

AD-A162 384

A GENERALIZED DBMS TO SUPPORT DIVERSE DATA(U)
CALIFORNIA UNIV BERKELEY ELECTRONICS RESEARCH LAB
L ROME ET AL 09 SEP 85 AFOSR-TR-85-0989 AFOSR-83-0254

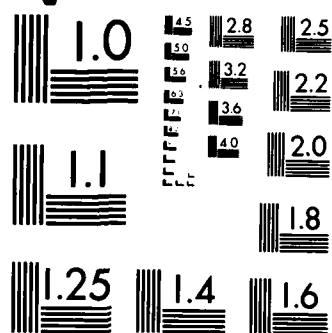
1/1

UNCLASSIFIED

F/G 9/2

NL

A 10x10 grid of 100 small images. The images show various patterns and textures, likely generated by a generative model. The patterns include solid colors, stripes, and abstract shapes. The colors are primarily black, white, and gray, with some images showing faint, blurry patterns. The grid is arranged in 10 rows and 10 columns.



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 35-0989	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Generalized DBMS to Support Diverse Data		5. TYPE OF REPORT & PERIOD COVERED Annual Scientific Report 1 July 1983 - 30 June 1984
7. AUTHOR(s) L. Rowe, M. Stonebraker and E. Wong		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION NAME AND ADDRESS Electronics Research Laboratory University of California Berkeley, CA 94720		8. CONTRACT OR GRANT NUMBER(s) AFOSR-83-0254
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research Bldg. 410, Bolling Air Force Base Washington, DC 20332		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 01102F 2304 A2
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE September 9, 1985
		13. NUMBER OF PAGES 82
		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) unlimited Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) DTIC ELECTE DEC 9 1985 A		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) INGRES, database, complex objects, extendible DBMS		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The authors proposed a research program to develop a generalized database manager to support diverse kinds of data including text, icons, forms, maps and other spatial data. The proposed research also included investigating support for integrated data browsers to allow end-users to query, step through, and update diverse data. Specific topics to be investigated included query language facilities to support text and geometric data, user-defined abstract data types in a DBMS, an ordered relation access method for text and other ordered data, extended secondary indexes, main memory databases, concurrency control, (cont on back)...		

AD-A162 384

DTIC FILE COPY

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

for data browsers, and an application program interface based on windows.

This paper reports on our progress during the first year of this program. The major advances have been made in the areas of abstract data types, main memory data bases and extended secondary indexes.

Project for	
Project	<input checked="" type="checkbox"/>
Project	<input type="checkbox"/>
Project	<input type="checkbox"/>
<i>See Supp Notes</i>	
<i>(all together)</i>	
<i>HP</i>	
Project Codes	
Date	and/or Special
<i>A-1</i>	



AFOSR-TR-

9

ANNUAL SCIENTIFIC REPORT
AIR FORCE OFFICE OF SCIENTIFIC RESEARCH

A GENERALIZED DBMS TO SUPPORT DIVERSE DATA

M. Stonebraker, L. Rowe and E. Wong
Principle Investigators

July 1, 1983 - June 30, 1984

Approved for public
distribution

ELECTRONICS RESEARCH LABORATORY
College of Engineering
University of California, Berkeley
94720

ABSTRACT

The authors proposed a research program to develop a generalized database manager to support diverse kinds of data including text, icons, forms, maps and other spatial data. The proposed research also included investigating support for integrated data browsers to allow end-users to query, step through, and update diverse data. Specific topics to be investigated included query language facilities to support text and geometric data, user-defined abstract data types in a DBMS, an ordered relation access method for text and other ordered data, extended secondary indexes, main memory databases, concurrency control for data browsers, and an application program interface based on windows.

RESEARCH RESULTS DURING THE 1983-1984 YEAR

We have made significant progress in our exploration of abstract data types, main memory data bases and extended secondary indexes during the last year. We comment briefly on significant accomplishments in each of these areas in the following subsections.

Abstract Data Types

One of our goals was to investigate ways of representing more complex objects such as geometric objects, text, maps, etc. in a relational data base system. Our major research result involves the possibility of using a query language to represent data types. This research is explored in detail in [STON84], which is included as an appendix to this document. Here we briefly review two previous proposals, then indicate the significance of our current contribution.

Our previous approach to supporting complex objects in a data base system was through abstract data types. We suggested allowing new types of columns to be added to a data base system along with new operators on these columns

[STONE83]. For example, a skilled programmer could define polygons as a new data type along with a collection of operators on data of type polygon including an intersection operator (!!). Then, other users could use polygons in the same way they use the built-in types of a data manager (floating point numbers, integers, and character strings). For example, a user could define the following POLYGON relation

```
create POLYGON (pid = i4, p-desc = polygon)
```

and then find the polygons overlapping the unit square as follows:

```
retrieve (POLYGON.all) where POLYGON.p-desc !! "0,0,1,1"
```

Support for user defined types and new operators has been constructed in about 2500 lines of code for the INGRES relational data base system. Implementation details are addressed in [FOGG82, ONG82], and ADTs run with a modest performance degradation [FOGG82]. Initial suggestions concerning how to integrate new operators into query processing heuristics and access methods are contained in [STON83, ONG84].

The abstract data type approach to complex objects is conceptually clean because no facilities peculiar to a specific kind of data are required. However, it has the disadvantage that one cannot easily "open up" an object and examine its component sub-objects. For example, suppose an airplane wing is defined as an abstract data type composed of a collection of components (e.g. cowlings, engines, etc.). In turn, each component could be composed of sub-components. Then, suppose a user wished to isolate a turbine blade inside a specific engine on a wing. To perform this task he would need two operators, one to "open up" the wing and identify a specific engine and a second to "open up" the engine and identify the desired turbine blade. Two cascaded operators are an awkward way to search in a complex object for a specific sub-object.

A second approach is to extend a relational data base system with specific facilities for particular complex objects. This is the approach taken in [LOR183] for objects appropriate for CAD applications. It has the advantage that component objects can be addressed but requires special-purpose services from a DBMS.

Our major contribution under the current grant is a third approach which may offer the good features of each of the above proposals. It involves supporting commands in the query language as a data type in a DBMS. In our environment this means that a column of a relation can have values which are one (or more) commands in the data manipulation language QUEL. We explain our proposal using the following example. Suppose a complex object is composed of lines, text and polygons. Each component is described in a separate relation as follows:

```
LINE (Lid, l-desc)
TEXT (Tid, t-desc)
POLYGON (Pid, p-desc)
```

Example inserts into the LINE and POLYGON relation are:

```
append to LINE (Lid = 22,
  description = "(0,0) (14,28)")
append to POLYGON (Pid = 44,
  description = "(1,10) (14,22) (6,19) (12,22)")
```

Then, the object as a whole would be stored in the OBJECT relation:

```
OBJECT (oid, o-desc)
```

The description field in OBJECT would be of type QUEL and contain queries to assemble the pieces of any given object from the other relations. For example, the following insert would make object 6 composed of line 22 and polygon 44.

```
append to OBJECT(
  oid = 6,
  o-desc = "retrieve (LINE.all) where LINE.id = 22
    retrieve (POLYGON.all) where pid = 44")
```


We have proposed extensions to QUEL which allow the components of an object to be addressed. For example, one could retrieve all the line descriptions making up object 6 which were of length greater than 10 as follows:

```
range of O is OBJECT
retrieve of (O.o-desc.1-desc) where
    length (O.o-desc.1-desc)>10
```

This notation has many points in common with the data manipulation language GEM [ZANI83], and allows one to conveniently discuss subsets of components of complex objects. In addition we can support clean sharing of lines, text and polygons among multiple composite objects by having the same query in the description field of more than one object, a feature lacking in the proposal of [LORI83].

Main Memory Data Bases

We have spent considerable time exploring the use of large amounts of main memory to speed data base processing. In [DEWI84] we have presented our results. The main contributions are:

- 1) an investigation of the viability of AVL trees in an environment where most of a relation may be present in main memory
- 2) an investigation of new algorithms for performing relational joins which can effectively utilize large quantities of main memory
- 3) an investigation of efficient means of obtaining crash recovery for a data base mostly resident in main memory

Our conclusions are somewhat counterintuitive. For example, it is found that merge-sort [SELI79] is rarely effective as a join tactic in an environment with much main memory because it is outperformed by several variations of hashing joins. A full discussion of this point and others appears in [DEWI84], which is also included in the appendix.

Extended Secondary Indices

During the past year we have concentrated on designing a secondary indexing structure that would be appropriate for solid geometric objects such as the polygons and rectangles which appear in CAD applications. Conventional spatial indexing schemes (e.g. KDB trees [ROBI81]) are only appropriate for point data. Our scheme, R-trees, allows efficient access to solid spatial objects according to their location [GUTT84]. Leaf nodes in an R-tree contain index entries, each consisting of a pointer to a spatial object and a rectangle that bounds the object. Higher nodes contain similar entries, with pointers to lower nodes and rectangles bounding the objects in the lower nodes. This hierarchy of covering rectangles is built and maintained dynamically in a manner similar to a B+tree.

To search for all data overlapping a given rectangle, we examine the root node to find which entries have rectangles overlapping the search area. The corresponding subtrees can have data in the search area, and therefore we apply the search algorithm recursively to each one. In this way we find all qualifying data, but avoid searching parts of the tree corresponding to objects that are far from the search area.

R-trees can be built for any number of dimensions. In addition they are useful for overlapping objects of non-zero size, a characteristic not shared by most multi-dimensional indexing schemes, for example quad trees [FINK74], k-d trees [BENT75], and K-D-B trees [ROBI81].

We have implemented R-trees, and in spatial search tests using VLSI data, only about 150 usec of CPU time was required to find each qualifying item. This indicates that the structure effectively restricts processing to qualifying or near-qualifying data. Our paper on R-trees [GUTT84] is included in the appendix to this proposal.

REFERENCES

- [BENT75] Bentley, J.L., "Multidimensional Binary Search Trees Used for Associative Searching," CACM 18,9 (September 1975), 509-517.
- [DEWI84] Dewitt, D., et. al., "Implementation Techniques for Main Memory Database Systems," to appear in Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1984.
- [FINK74] Finkel, R.A. and J.L. Bentley, "Quad Trees - A Data Structure for Retrieval on Composite Keys," Acta Informatica 4, (1974), 1-9.
- [FOGG82] Fogg, D., "Implementation of Domain Abstraction in the Relational Database System, INGRES," Masters Report, EECS Dept, University of California, Berkeley, Sept. 1982.
- [GUTT84] Guttman, A., "R-Trees: A Dynamic Index Structure for Spatial Searching," to appear in 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1984.
- [LORI83] Lorie, R. and W. Plouffe, "Complex Objects and Their Use in Design Transactions," Proc. Engineering Design Applications of ACM-IEEE Data Base Week, San Jose, Ca., May 1983.
- [ONG82] Ong, J., "The Design and Implementation of Abstract Data Types in the Relational Database System, INGRES," Masters Report, EECS Dept, University of California, Berkeley, Sept. 1982.
- [ONG84] Ong, J., et. al., "Implementation of Data Abstraction in the Relational Database System INGRES," to appear in SIGMOD Record.
- [ROBI81] Robinson, J.T., "The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes," Proc 1981 ACM-SIGMOD Conference, Ann Arbor, Mich., June 1981.
- [SELI79] Selinger, PP., et. al., "Access Path Selection in a Relational Data Base Management System," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1979.
- [STON83] Stonebraker, M. et. al., "Application of Abstract Data Types and Abstract Indices to CAD Databases," Proc. Engineering Design Applications of ACM-IEEE Database Week, San Jose, Ca., May 1983.
- [STON84] Stonebraker, M., et. al., "QUEL as a Data Type," to appear in Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1984.
- [ZANI83] Zaniola, C., "The Database Language GEM," "Proc. 1983 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., May 1983.

PROFESSIONAL PERSONNEL

Michael Stonebraker, PI

Erika Anderson: B.A. (Computer Science), May 1984.

Antonin Gutman: PhD August 1984, "Application of Relational Data Base Systems to CAD Data"

John Woodfill: B.S. (Computer Science), May 1984.

PUBLICATION CITATIONS

David J. Dewitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker and David Wood, "Implementation Techniques for Main Memory Database Systems," *Proc. 1984 ACM-SIGMOD Conference on Management of Data*, Boston, Mass., June 1984.

Michael Stonebraker, Erika Anderson, Eric Hansen and Brad Rubinstein, "QUEL as a Data Type," *Proc. 1984 ACM-SIGMOD Conference on Management of Data*, Boston, Mass., June 1984.

A. Gutman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. 1984 ACM-SIGMOD Conference on Management of Data*, Boston, Mass., June 1984.

Michael Stonebraker, "Virtual Memory Transaction Management," *ACM SIGOPS Review*, April 1984.

Michael Stonebraker, John Woodfill and Erika Anderson, "Implementation of Rules in Relational Data Base Systems," *ACM SIGMOD Record*, September 1983.

INTERACTIONS

The first three papers in the Publication Citations section were presented at ACM SIGMOD annual conference.

A presentation on "QUEL as a Data Type" was given at Wang Institute and IBM Federal Systems Division.

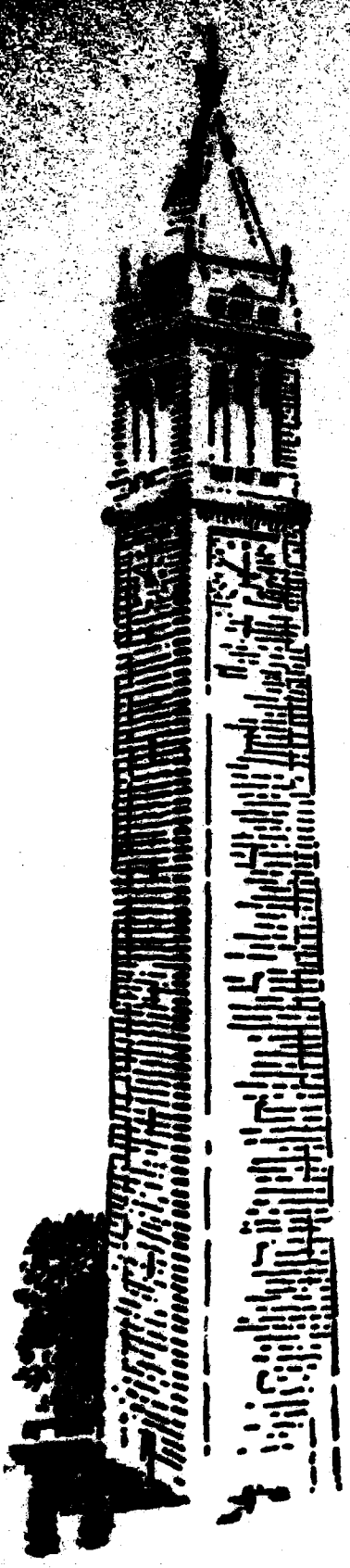
R-TREES: A DYNAMIC INDEX STRUCTURE
FOR SPATIAL SEARCHING

by

Antonin Guttman and Michael Stonebraker

Memorandum No. UCB/ERL M83/64

14 October 1983



ELECTRONICS RESEARCH LABORATORY
College of Engineering 85 12 6 047
University of California, Berkeley, CA 94720

R-TREES: A DYNAMIC INDEX STRUCTURE
FOR SPATIAL SEARCHING

by

Antonin Guttman and Michael Stonebraker

Memorandum No. UCB/ERL M83/64

14 October 1983

ELECTRONICS RESEARCH LABORATORY
College of Engineering
University of California, Berkeley
94720

Research sponsored by the National Science Foundation Grant ECS-8300463
and the Air Forces Office of Scientific Research Grant AFOSR-83-0254.

1. Introduction

Spatial data objects often cover areas in multi-dimensional spaces and are not well represented by point locations. For example, map objects like counties, census tracts etc. occupy regions of non-zero size in two dimensions. A common operation on spatial data is to search for all objects in an area. An example would be to find all the counties that have land within 20 miles of a particular point. This kind of spatial search occurs frequently in computer aided design (CAD) and geo-data applications. In such applications it is important to be able to retrieve objects efficiently according to their spatial location.

An index based on objects' spatial locations is desirable, but classical one-dimensional database indexing structures are not appropriate to multi-dimensional spatial searching. Structures based on exact matching of values, such as hash tables, are not useful because a range search is required. Structures using one-dimensional ordering of key values, such as B-trees and ISAM indexes, do not work because the search space is multi-dimensional.

A number of structures have been proposed for handling multi-dimensional point data, and a survey of methods can be found in [5]. Cell methods [4, 8, 16] are not good for dynamic structures because the cell boundaries must be decided in advance. Quad trees [7] and k-d trees [3] do not take paging of secondary memory into account. K-D-B trees [13] are designed for paged memory but are only useful for point data. The use of index intervals has been suggested in [15], but this method cannot be used in multiple dimensions. Corner stitching [12] is an example of a structure for two-dimensional spatial searching suitable for data objects of non-

zero size, but it assumes homogeneous primary memory and is not efficient for random searches in very large collections of data. Grid files [10] handle non-point data by mapping each object to a point in a higher-dimensional space. In this paper we describe an alternative structure called an R-tree which represents data objects by intervals in several dimensions.

Section 2 outlines the structure of an R-tree and Section 3 gives algorithms for searching, inserting, deleting, and updating. Results of R-tree index performance tests are presented in Section 4. Section 5 contains a summary of our conclusions.

2. R-Tree Index Structure

An R-tree is an index structure for n-dimensional spatial objects analogous to a B-tree [2,6]. It is a height-balanced tree with records in the leaf nodes each containing an n-dimensional rectangle and a pointer to a data object having the rectangle as a bounding box. Higher level nodes contain similar entries with links to lower nodes. Nodes correspond to disk pages if the structure is disk-resident, and the tree is designed so that a small number of nodes will be visited during a spatial search. The index is completely dynamic; inserts and deletes can be intermixed with searches and no periodic reorganization is required.

A spatial database consists of a collection of records representing spatial objects, and each record has a unique identifier which can be used to retrieve it. We approximate each spatial object by a bounding rectangle, i.e. a collection of intervals, one along each dimension:

$$I = (I_0, I_1, \dots, I_{n-1})$$

where n is the number of dimensions and I_i is a closed bounded interval $[a, b]$ describing the extent of the object along dimension i . Alternatively I_i may have one or both endpoints equal to infinity, indicating that the object extends outward indefinitely.

Leaf nodes in the tree contain index record entries of the form

$$(I, \text{tuple-identifier})$$

where *tuple-identifier* refers to a tuple in the database and I is an n -dimensional rectangle containing the spatial object it represents. Non-leaf nodes contain entries of the form

$$(I, \text{child-pointer})$$

where *child-pointer* is the address of another node in the tree and I covers all rectangles in the lower node's entries. In other words, I spatially contains all data objects indexed in the subtree rooted at I 's entry.

Let M be the maximum number of entries that will fit in one node and let $m \leq \frac{M}{2}$ be a parameter specifying the minimum number of entries in a node. An R-tree satisfies the following properties:

- (1) Every leaf node contains between m and M index records unless it is the root.
- (2) For each index record $(I, \text{tuple-identifier})$ in a leaf node, I is the smallest rectangle that spatially contains the n -dimensional data object represented by the indicated tuple.
- (3) Every non-leaf node has between m and M children unless it is the root.
- (4) For each entry $(I, \text{child-pointer})$ in a non-leaf node, I is the smallest rectangle that spatially contains the rectangles in the child node.

- (5) The root node has at least two children unless it is a leaf.
- (6) All leaves appear on the same level.

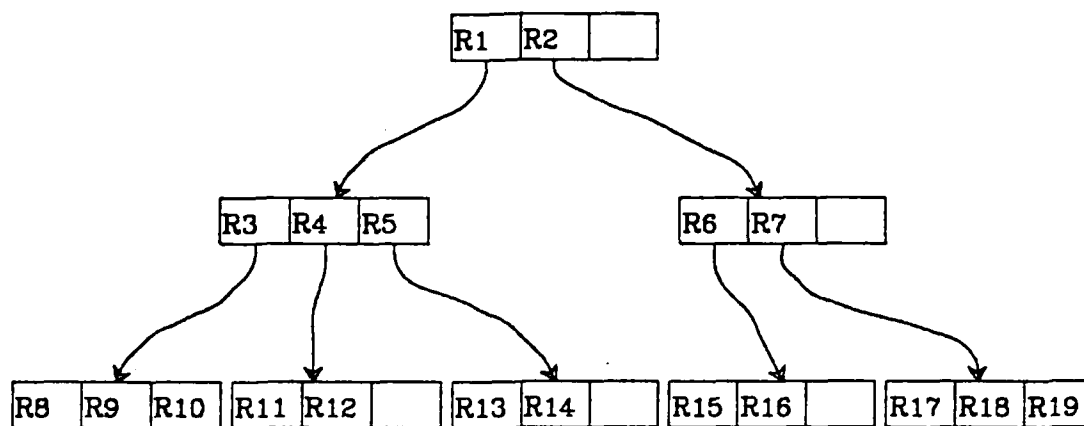
Figure 2.1a and 2.1b show an example R-tree structure and the geometric forms it represents.

The height of an R-tree containing N index records is at most $\lceil \log_m N \rceil$, because the branching factor of each node is at least m . The maximum number of nodes is $\lceil \frac{N}{m} \rceil + \lceil \frac{N}{m^2} \rceil + \dots + 1$. Worst-case space utilization for all nodes except the root is $\frac{m}{M}$. Nodes will tend to have more than m entries, and this will decrease tree height and improve space utilization. If nodes have more than 3 or 4 entries the tree will be very wide, and almost all the space will be used for leaf nodes containing index records. The parameter m can be varied as part of performance tuning, and different values are tested experimentally in Section 4.

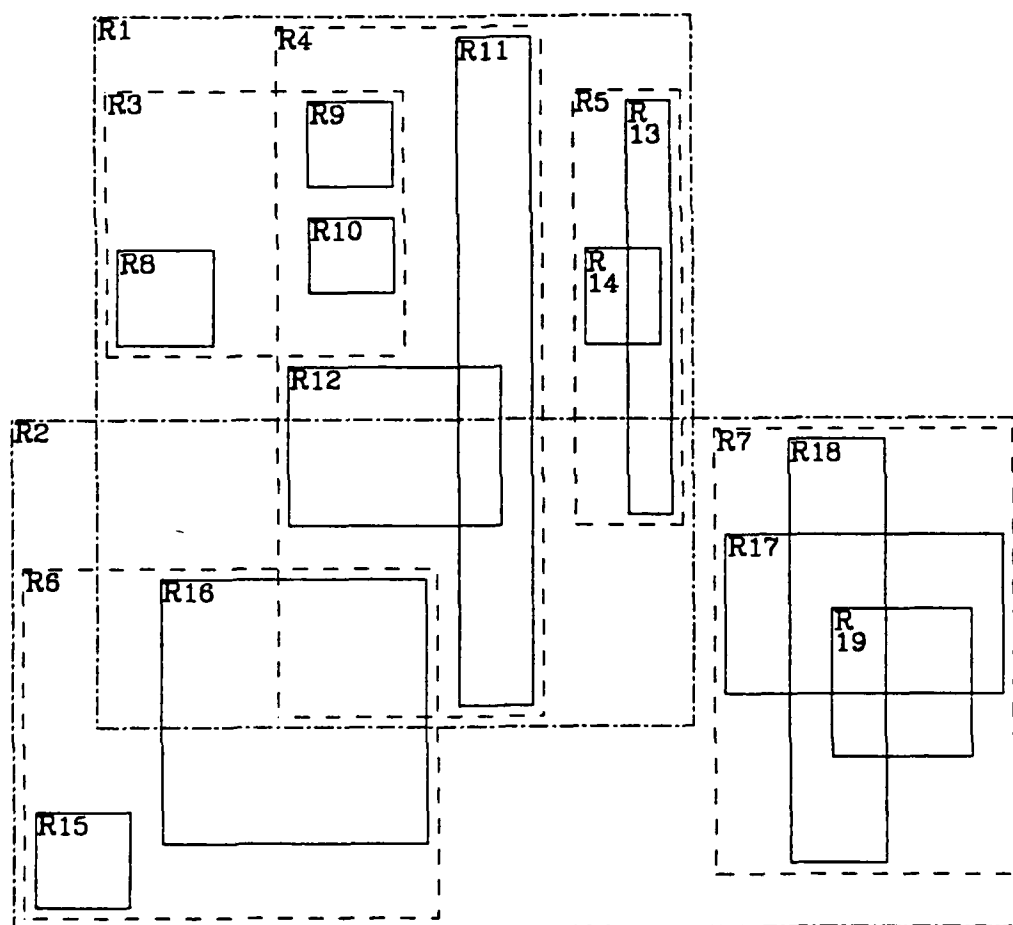
3. Searching and Updating

3.1. Searching

The search algorithm descends the tree from the root in a manner similar to a B-tree. However when it visits a non-leaf node it may find that any number of subtrees from 0 to M need to be searched; hence it is not possible to guarantee good worst-case performance. Nevertheless with most kinds of data the update algorithms will maintain the tree in a form that allows the search algorithm to eliminate irrelevant regions of the indexed space, and examine only data near the search area.



(a)



(b)

Figure 2.1

In the following we denote the rectangle part of an index entry E by $E.I$, and the *tuple-identifier* or *child-pointer* part by $E.p$.

Algorithm Search. Given an R-tree whose root node is T , find all index records whose rectangles overlap a search rectangle S .

S1. [Search subtrees.]

If T is not a leaf, check each entry E to determine whether $E.I$ overlaps S . For all overlapping entries, invoke **Search** on the tree whose root node is pointed to by $E.p$.

S2. [Search leaf node.]

If T is a leaf, check all entries E to determine whether $E.I$ overlaps S . If so, E is a qualifying record.

3.2. Insertion

Inserting index records for new data tuples is similar to insertion in a B-tree in that new index records are added to the leaves, nodes that overflow are split, and splits propagate up the tree.

Algorithm Insert. Insert a new index entry E into an R-tree.

I1. [Find position for new record.]

Invoke **ChooseLeaf** to select a leaf node L in which to place E .

I2. [Add record to leaf node.]

If L has room for another entry, install E . Otherwise invoke **SplitNode** to obtain L and LL containing E and all the old entries of L .

13. [Propagate changes upward.]

Invoke **ExpandTree** on L , also passing LL if a split was performed.

14. [Grow tree taller.]

If node split propagation resulted in the root being split, create a new root whose children are the two nodes resulting from the split.

Algorithm ChooseLeaf. Select a leaf node of an R-tree in which to place a new index entry E .

CL1. [Initialize.]

Set N to be the root node.

CL2. [Leaf check.]

If N is a leaf, return N .

CL3. [Choose subtree.]

If N is not a leaf, let F be the entry in N whose rectangle $F.I$ needs least enlargement to include $E.I$. Resolve ties by choosing the entry with the rectangle of smallest area.

CL4. [Descend until a leaf is reached.]

Set N to be the child node pointed to by $F.p$ and repeat from CL2.

Algorithm ExpandTree. Ascend from a leaf node L in an R-tree to the root, adjusting covering rectangles and propagating node splits as necessary.

ET1. [Initialize.]

Set $N=L$. If L was split previously, set NN to be the resulting second node.

ET2. [Check if done.]

If N is the root, stop.

ET3. [Adjust covering rectangle in parent entry.]

Let P be the parent node of N , and let E_N be N 's entry in P . Adjust $E_N.I$ so that it tightly encloses all entry rectangles in N .

ET4. [Propagate node split upward.]

If N has a partner NN resulting from an earlier split, create a new entry E_{NN} with $E_{NN}.p$ pointing to NN and $E_{NN}.I$ enclosing all rectangles in NN . Add E_{NN} to P if there is room. Otherwise, invoke **SplitNode** to produce P and PP containing E_{NN} and all P 's old entries.

ET5. [Move up to next level.]

Set $N=P$ and set $NN=PP$ if a split occurred. Repeat from ET2.

Algorithm **SplitNode** is described in Section 3.5.

3.3. Deletion

Algorithm **Delete**. Remove index record E from an R-tree.

D1. [Find node containing record.]

Invoke **FindLeaf** to locate the leaf node L containing E . Stop if the record was not found.

D2. [Delete record.]

Remove E from L .

D3. [Adjust tree.]

Invoke **CondenseTree** to adjust the covering rectangles on the path from L to the root, to eliminate under-full nodes, and to propagate node eliminations up the tree.

D4. [Shorten tree.]

If the root node has only one child after the tree has been adjusted, make the child the new root.

Algorithm FindLeaf. Given an R-tree whose root node is T , find the leaf node containing the index entry E .

FL1. [Search subtrees.]

If T is not a leaf, check each entry F in T to determine if $F.I$ overlaps $E.I$. For each such entry invoke **FindLeaf** on the tree whose root is pointed to by $F.p$ until E is found or all entries have been checked.

FL2. [Search leaf node for record.]

If T is a leaf, check each entry to see if it matches E . If E is found return T .

Algorithm CondenseTree. Given an R-tree leaf node L from which an entry has been deleted, eliminate the node if it has too few entries and relocate its entries. Propagate node elimination upward as necessary. Adjust all covering rectangles on the path to the root, making them smaller if possible.

CT1. [Initialize.]

Set $N=L$. Set Q , the set of eliminated nodes, to be empty.

CT2. [Find parent entry unless root has been reached.]

If N is the root, go to CT6. Otherwise let P be the parent of N , and let E_N be N 's entry in P .

CT3. [Eliminate under-full node.]

If N has fewer than m entries, delete E_N from P and add N to set Q .

CT4. [Adjust covering rectangle.]

If N has not been eliminated, adjust $E_N.I$ to tightly contain all entries in N .

CT5. [Move up one level in tree.]

Set $N=P$ and repeat from CT2.

CT6. [Re-insert orphaned entries.]

Re-insert all entries of nodes in set Q . Entries from eliminated leaf nodes are re-inserted in tree leaves as described in Algorithm Insert, but entries from higher-level nodes must be placed higher in the tree. This is done so that leaves of their dependent subtrees will be on the same level as leaves of the main tree.

The procedure outlined above for disposing of under-full nodes differs from the corresponding operation on a B-tree, in which two or more adjacent nodes are merged. A B-tree-like approach is possible for R-trees, although there is no adjacency in the B-tree sense: an under-full node can be merged with whichever sibling will have its area increased least, or the orphaned entries can be distributed among sibling nodes. Either method can cause nodes to be split.

Re-insertion was chosen instead for two reasons: first, it accomplishes the same thing and is easier to implement because the `Insert` routine can be used. Efficiency should be comparable because pages needed during re-insertion usually will be the same ones visited during the preceding search and will already be in memory. The second reason is that re-insertion incrementally refines the spatial structure of the tree, and prevents gradual deterioration that might occur if each entry were located permanently under the same parent node.

3.4. Updates and Other Operations

If a data tuple is updated so that its covering rectangle is changed, its index record must be deleted, updated, and then re-inserted, so that it will find its way to the right place in the tree.

Other kinds of searches besides the one described above may be useful, for example to find all data objects completely contained in a search area, or all objects that contain a search area. These operations can be implemented by straightforward variations on the algorithm given. A search for a specific entry whose identity is known beforehand is required by the deletion algorithm and is implemented by Algorithm `FindLeaf`. Variants of range deletion, in which index entries for all data objects in a particular area are removed, are also well supported by R-trees.

3.5. Node Splitting

When an attempt is made to add an entry to a full node containing M entries, the collection of $M+1$ entries must be divided between two nodes. The division

should be done in a way that makes it as unlikely as possible that both new nodes will need to be examined on subsequent searches. Since the decision to visit a node is based on whether the search area overlaps the covering rectangle for the node's entries, the total area of the two covering rectangles should be minimized. Figure 3.1 illustrates this point. The area of the covering rectangles in the "bad split" case is much larger than in the "good split" case.

Note that the same criterion was used in procedure ChooseLeaf to decide where to insert a new index entry: at each level in the tree, the subtree was chosen whose covering rectangle would have to be enlarged least.

We now turn to algorithms for partitioning the set of $M+1$ entries into two groups, one for each new page.

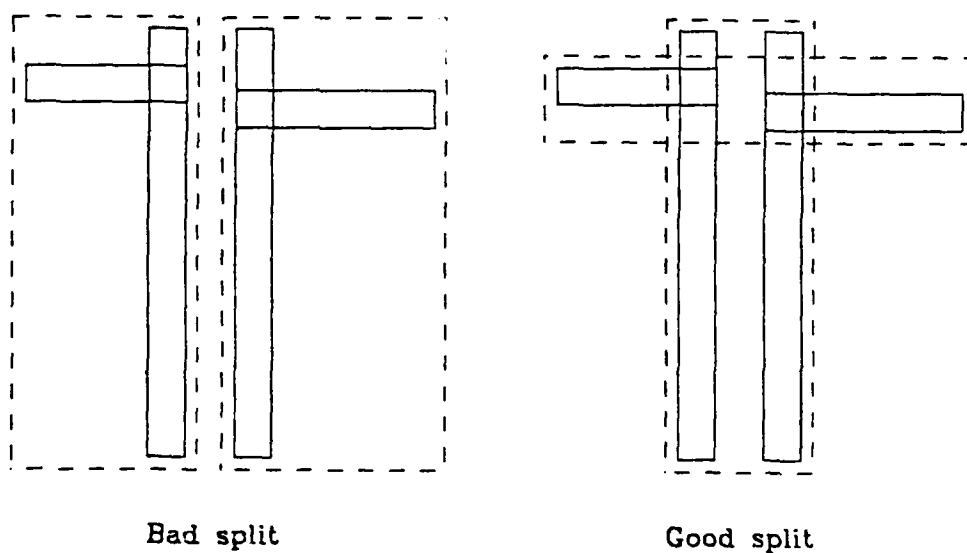


Figure 3.1

3.5.1. Exhaustive Algorithm

The most straightforward way to find the minimum area node split is to generate all alternatives and choose the best. However, the number of possible partitions is approximately 2^{M-1} and a reasonable value of M is 50*, so the number of possible splits is very large. We implemented a modified form of the exhaustive node split algorithm to use as a standard for comparison with other algorithms, but it was too slow to use with large page sizes.

3.5.2. A Quadratic-Cost Algorithm

This algorithm attempts to find a small-area split, but is not guaranteed to find one with the smallest area possible. The cost is quadratic in M and linear in the number of dimensions. The algorithm picks two of the $M+1$ entries to be the first elements of the groups which will make up the new nodes. The pair chosen is the one that would waste the most area if both entries were put in the same group, i.e. the area of the rectangle covering both entries, minus the areas of the rectangles in the entries, would be greatest. Then the remaining entries are selected one at a time and assigned to a group. At each step the area expansion required to add each entry to each group is calculated, and the entry chosen is the one showing the greatest difference between the two groups in the expansion required to include it.

Algorithm Quadratic Split. Divide a set of $M+1$ index entries into two groups.

* A two dimensional rectangle can be represented by four numbers of four bytes each. If a pointer also takes four bytes, each entry requires 20 bytes. A page of 1024 bytes will hold about 50 entries.

QS1. [Pick first entry for each group.]

Apply Algorithm **PickSeeds** to choose two entries to be the first elements of the groups. Assign each to a group.

QS2. [Check if done.]

If all entries have been assigned, stop. If one group has so few entries that all the rest must be assigned to it in order for it to have the minimum number m , assign them and stop.

QS3. [Select entry to assign.]

Invoke Algorithm **PickNext** to choose the next entry to assign. Add it to the group whose covering rectangle will have to be enlarged least to accommodate it. Resolve ties by adding the entry to the group with smaller area, then to the one with fewer entries, then to either. Repeat from QS2.

Algorithm PickSeeds. Select two entries to be the first elements of the groups.

PS1. [Calculate inefficiency of grouping entries together.]

For each pair of entries E_1 and E_2 , compose a rectangle J including $E_1.I$ and $E_2.I$. Calculate $d = \text{area}(J) - \text{area}(E_1.I) - \text{area}(E_2.I)$.

PS2. [Choose the most wasteful pair.]

Choose the pair with the largest d to be put in different groups.

Algorithm PickNext. Select one remaining entry for classification in a group.

PN1. [Determine cost of putting each entry in each group.]

For each entry E not yet in a group, calculate d_1 = the area increase required in the covering rectangle of Group 1 to include $E.I$. Calculate d_2 similarly for

Group 2.

PN2. [Find entry with greatest preference for one group.]

Choose the entry with the greatest difference between d_1 and d_2 . If more than one has the same lowest difference pick any of them.

3.5.3. A Linear-Cost Algorithm

This algorithm is linear in M and in the number of dimensions. First it selects seed entries for the two groups, choosing the two whose rectangles are the most widely separated along any dimension. Then it processes the remaining entries without ordering them in any special way, placing each in one of the groups.

Algorithm *Linear Split* is identical to *Quadratic Split* but uses a different version of *PickSeeds*. *PickNext* simply chooses any of the remaining entries.

Algorithm *LinearPickSeeds*. Select two entries to be the first elements of the groups.

LPS1. [Find extreme rectangles along all dimensions.]

Along each dimension, find the entry whose rectangle has the highest low side, and the one with the lowest high side. Record the separation.

LPS2. [Adjust for shape of the rectangle cluster.]

Normalize the separations by dividing by the width of the entire set along the corresponding dimension.

LPS3. [Select the most extreme pair.]

Choose the pair with the greatest normalized separation along any dimension.

4. Performance Tests

We implemented R-trees in C under Unix on a Vax 11/780 computer. Our implementation has been used for a series of performance tests, whose purpose was to verify the practicality of the structure, to choose values for M and m , and to evaluate different node-splitting algorithms. This section presents the results.

Five page sizes were tested, corresponding to different values of M :

Bytes per Page	Max Entries per Page (M)
128	6
256	12
512	25
1024	50
2048	102

The minimum number of entries in a node (m) was tested for values $M/2$, $M/3$, and 2. The three node split algorithms described earlier were implemented in different versions of the program.

All our tests used two-dimensional data, although the structure and algorithms work for any number of dimensions. During the first part of each test run the program read geometry data from files and constructed an index tree, beginning with an empty tree and calling *Insert* with each new index record. Insert performance was measured for the last 10% of the records, when the tree was nearly its final size.

During the second phase the program called the function *Search* with random search rectangles made up using the local random number generation facility. 100 searches were performed per test run, each retrieving about 5% of the data.

Finally the program read the input files a second time and called the function *Delete* to remove the index record for every tenth data item. Thus measurements

were taken for scattered deletion of 10% of the index records.

The tests were done using Very Large Scale Integrated circuit (VLSI) layout data from the RISC-II computer chip. [11]. The first series of tests used the circuit cell CENTRAL, containing 1057 rectangles (Figure 4.1).

Figure 4.2 shows the cost in CPU time for inserting the last 10% of the records as a function of page size. The exhaustive algorithm, whose cost increases exponentially with page size, is seen to be very slow for larger page sizes. The linear algorithm is fastest, as expected. With this algorithm CPU time hardly increased with page size at all, which suggests that node splitting was responsible for only a small

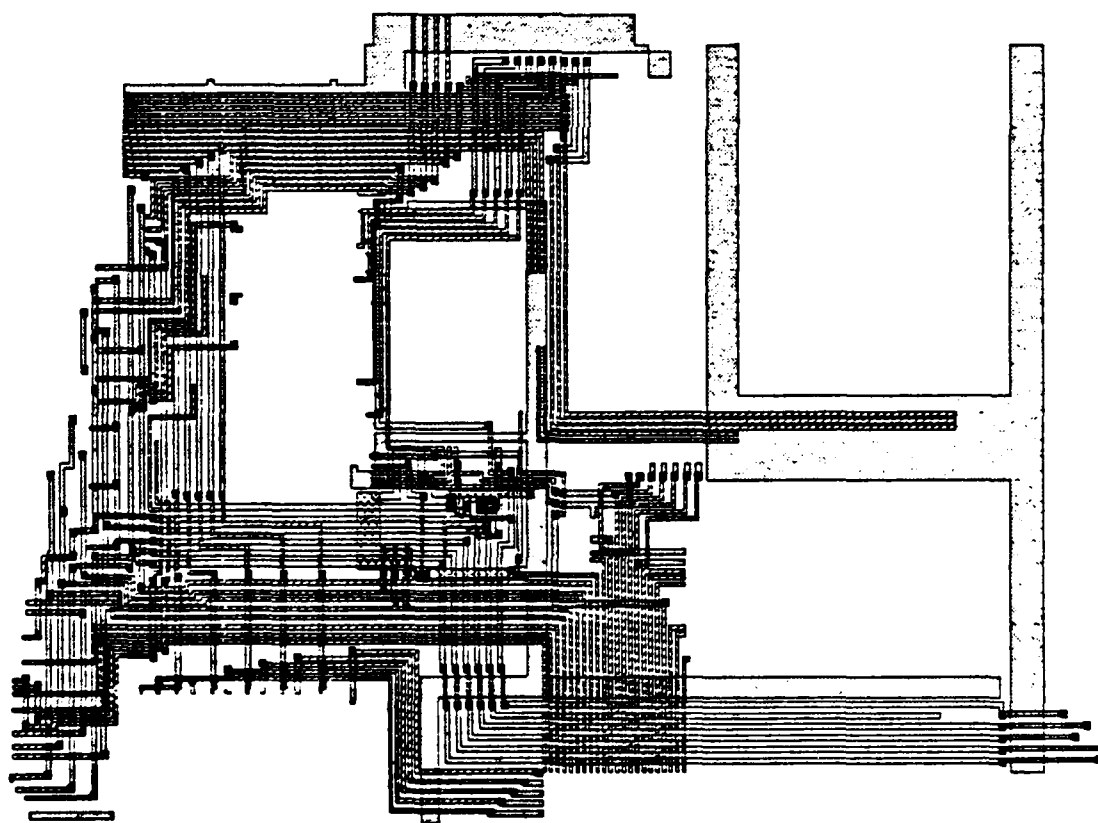


Figure 4.1
Circuit cell CENTRAL (1057 rectangles).

part of the cost of inserting records.

The decreased cost of the quadratic algorithm with a stricter node fill requirement reflects the fact that it stops doing expensive processing when one group becomes too full, and changes to a linear mode for the rest of the split.

The cost of deleting an item from the index, shown in Figure 4.3, is affected by the split algorithm and especially by the minimum node fill requirement. This is true because deletion indirectly causes node splitting when entries from under-full nodes are re-inserted. Stricter fill requirements cause nodes to be eliminated more frequently, and each one has more entries to re-insert. When the minimum fill is 2 re-insertion is rare; the cost of scanning nodes and traversing the tree dominates and

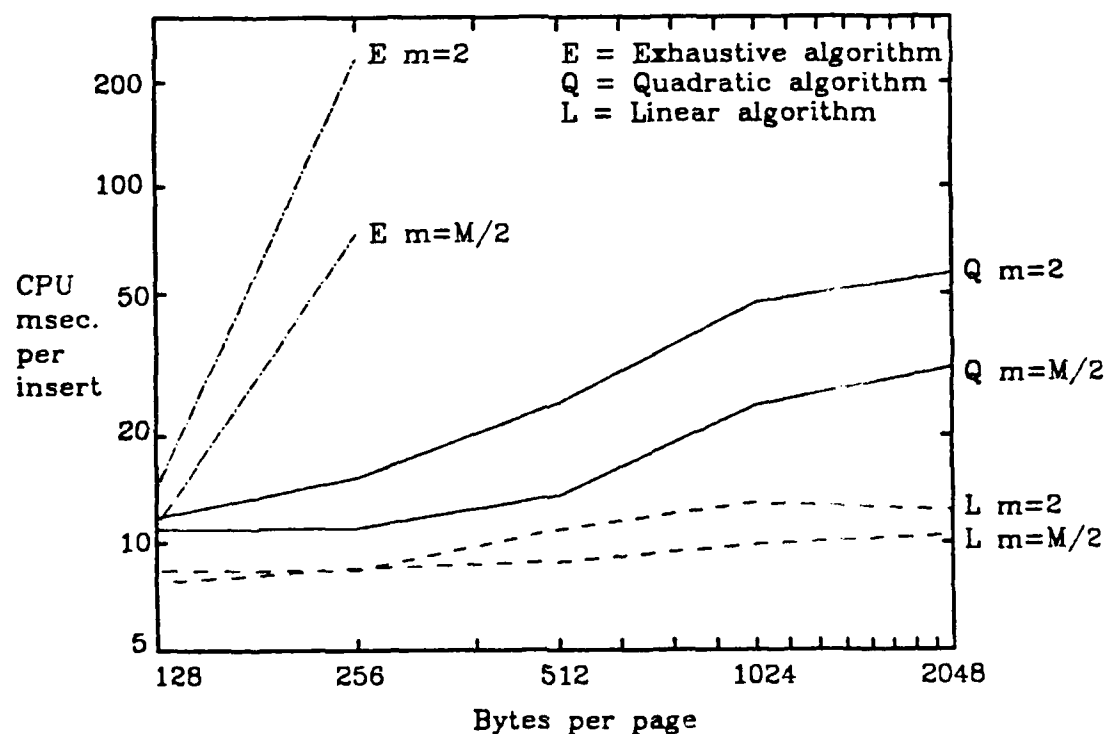


Figure 4.2
CPU cost of inserting records.

the split algorithm has little effect. The curves are rough because node eliminations occur randomly and infrequently; there were too few of them in our tests to smooth out the variations.

Figures 4.4 and 4.5 show that the search performance of the index is very insensitive to the use of different node split algorithms and fill requirements. The exhaustive algorithm produces a slightly better index structure, resulting in fewer pages touched and less CPU cost, but even the crudest algorithm, the linear one with $m=M/2$, provides reasonable performance. Most combinations of algorithm and minimum fill come within 10% of the performance achieved with node splits produced by the exhaustive algorithm.

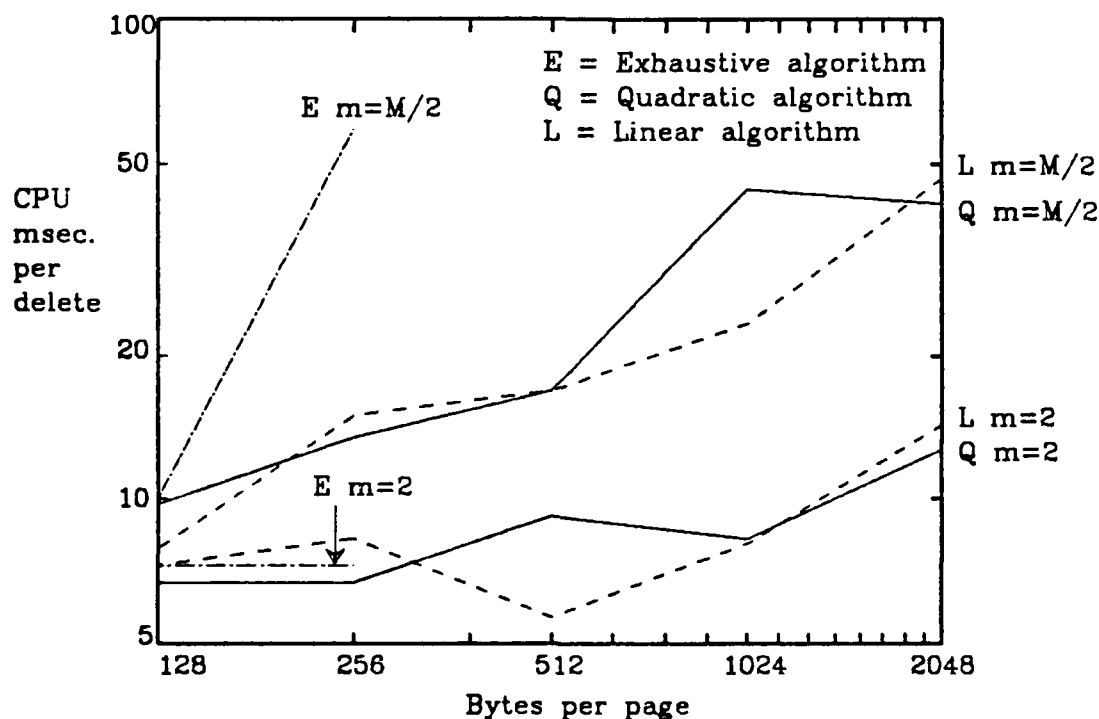


Figure 4.3
CPU cost of deleting records.

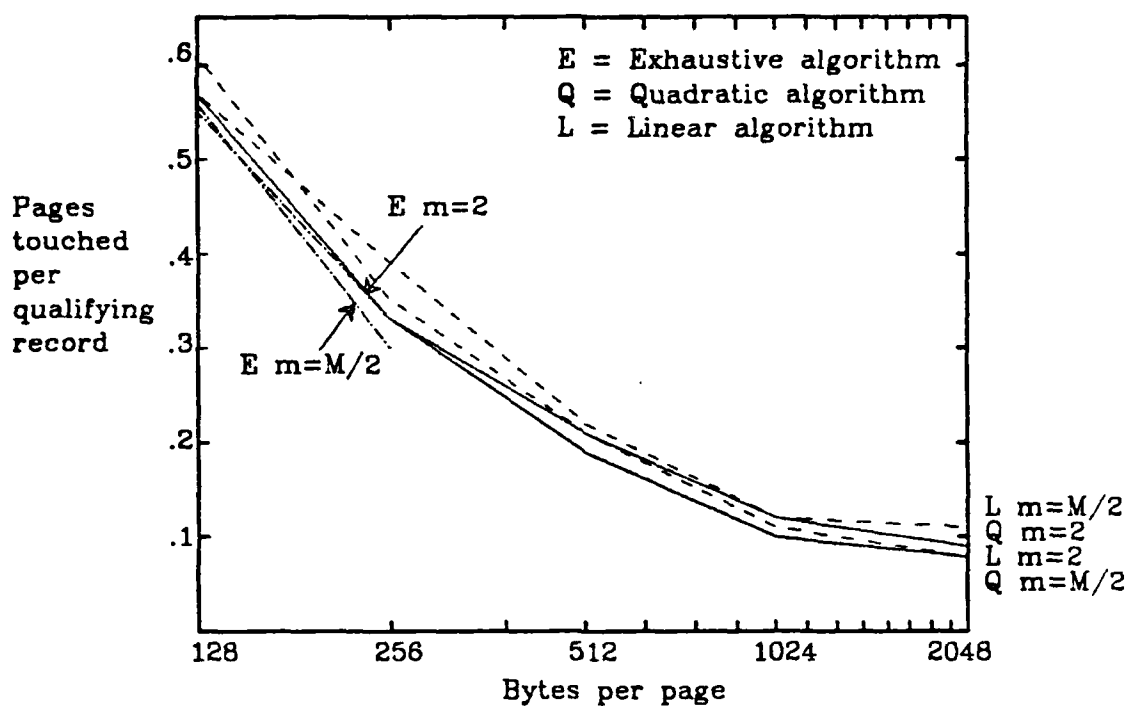


Figure 4.4
Search performance: Pages touched.

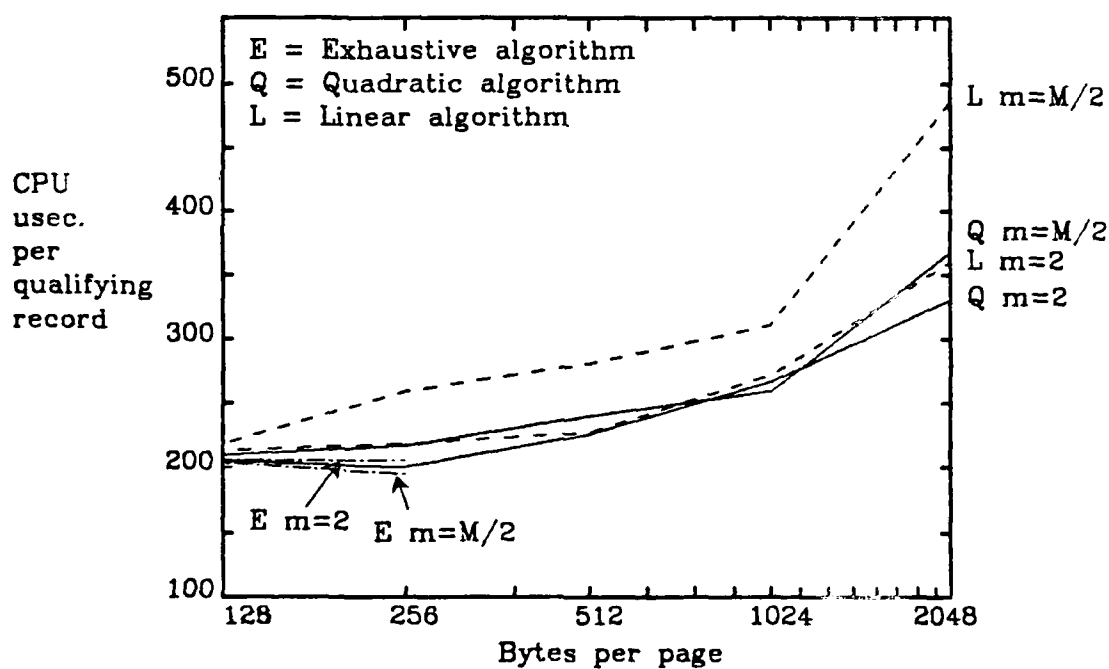


Figure 4.5
Search performance: CPU cost.

Figure 4.6 shows the storage space occupied by the index tree as a function of algorithm, fill criterion and page size. Generally the results bear out our expectation that stricter node fill criteria produce smaller indexes. The least dense index consumes about 50% more space than the most dense, but all results for 1/2-full and 1/3-full (not shown) are within 15% of each other.

We performed a second series of tests to measure R-tree performance as a function of the amount of data in the index. The same sequence of test operations as before was run on samples containing 1057, 2238, 3295, and 4559 rectangles. The first sample contained layout data from the same circuit cell CENTRAL as used earlier. The second sample consisted of layout from a larger circuit cell containing 2238 rectangles. The third sample was made by using both CENTRAL and the larger

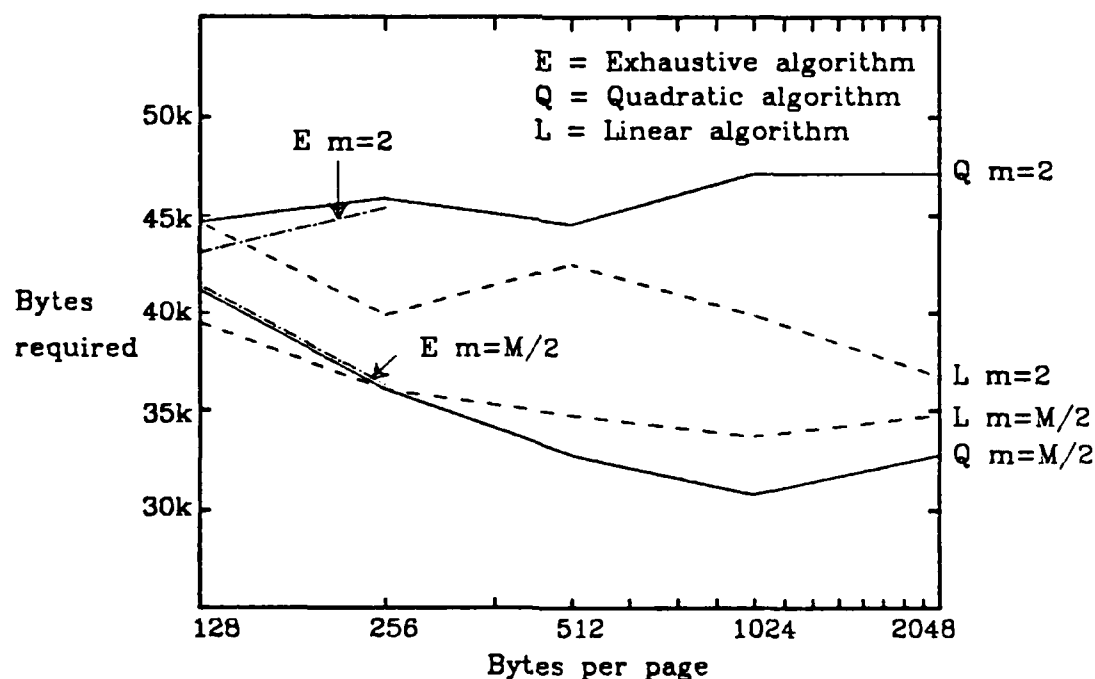


Figure 4.6
Space efficiency.

cell, with the two cells effectively placed on top of each other. Three cells were combined to make up the last sample. Because the samples were all composed in different ways using varying data, performance results would not scale perfectly and some unevenness was to be expected.

Two combinations of algorithm and node fill requirement were chosen for the tests: the linear split algorithm with $m=2$, and the quadratic algorithm with $m=M/3$, both with a page size of 1024 bytes ($M=50$). Both have good search performance; the linear configuration has about half the insert cost, and the quadratic one produces a smaller index.

Figure 4.7 shows the results of tests whose purpose was to determine how insert and delete performance is affected by tree size. Both test configurations produced trees with two levels for 1057 records and three levels for the other sample sizes. The figure shows that the cost of inserts with the quadratic algorithm is nearly constant except where the tree increases in height, where the curve shows a definite jump because the number of levels where an expensive split can occur increases. The linear algorithm shows no jump, indicating again that linear node splits account for only a small part of the cost of inserts.

Deletion with the linear configuration has nearly constant cost for fixed tree height, and a small jump where the tree becomes taller. The tests involved too few deletions to cause any node splits on re-insertion with the relaxed node fill requirement. A few splits did occur during deletion with the quadratic configuration, but only 1 to 6 per test run, and the number of nodes eliminated was similarly small, varying between 4 and 15. Such small numbers, coupled with the high cost of node

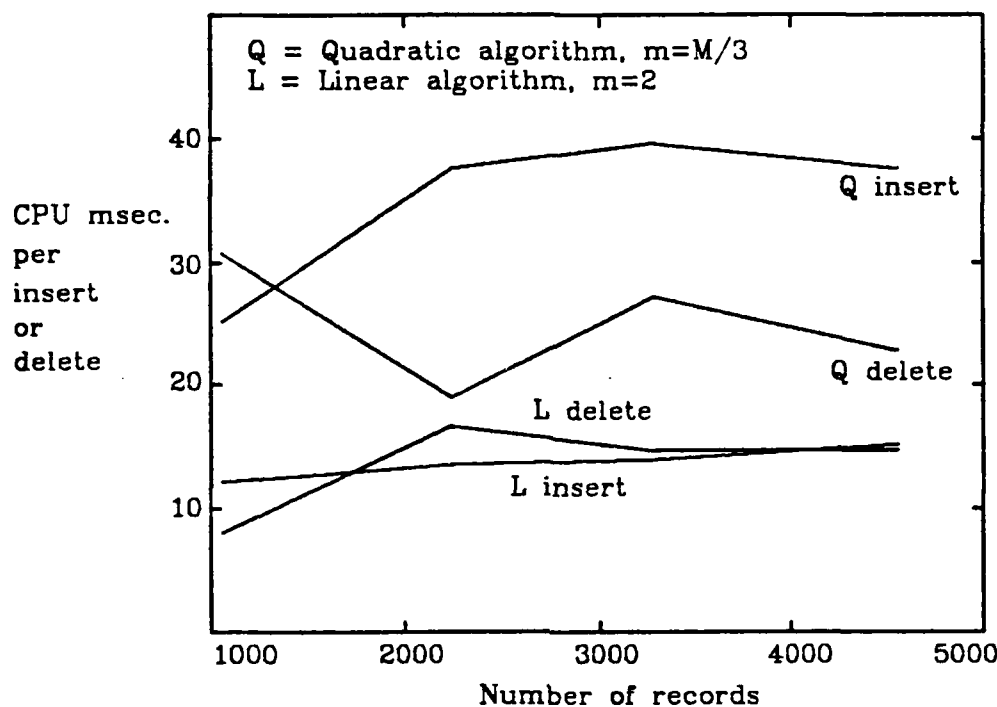


Figure 4.7
CPU cost of inserts and deletes vs. amount of data.

eliminations, made the quadratic curve very erratic.

When allowance is made for variations due to the small sample size, the tests show that insert and delete cost is independent of tree width but is affected by tree height, which grows slowly with the number of data items. This performance is very similar to that of a B-tree.

The search tests retrieved between 3% and 6% of the data on each search. Figures 4.8 and 4.9 confirm that the two configurations have nearly the same performance. The downward trend of the curves is to be expected, because the cost of processing higher tree nodes becomes less significant as the amount of data retrieved in each search increases. The decrease would have been more uniform, but an unexpected variation in the 2238-item sample raised the cost at the second data point.

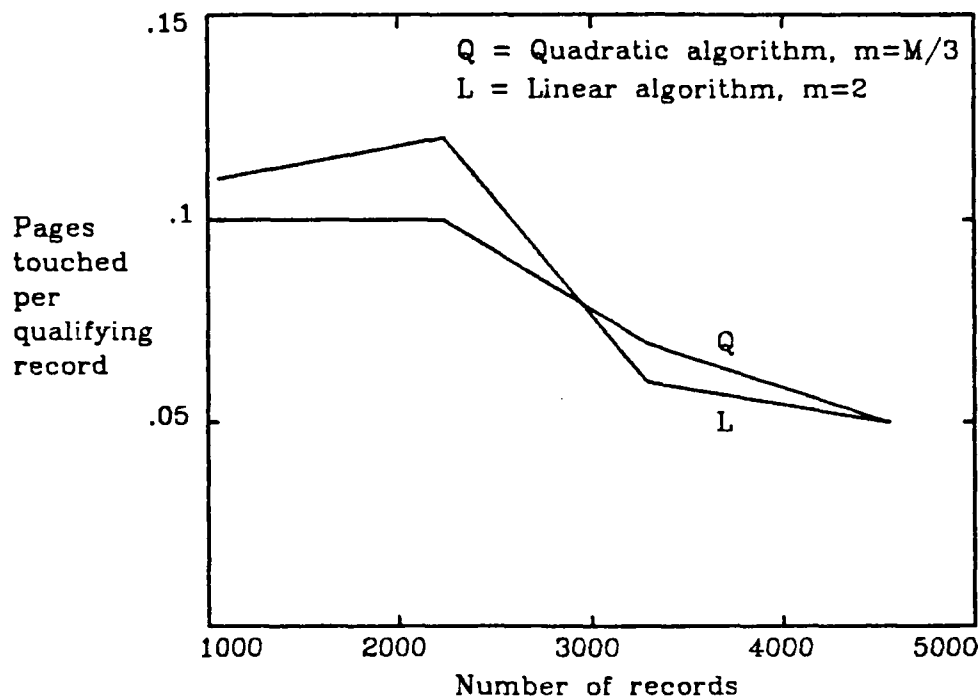


Figure 4.8
Search performance vs. amount of data: Pages touched

This sample was considerably denser than the others and contained more long shapes spanning many small ones. The search rectangles tended to overlap a larger fraction of the bins in the index, which resulted in less efficient searching.

The low CPU cost per qualifying record, less than 150 microseconds for larger amounts of data, shows that the index is quite effective in narrowing searches to small subtrees.

The straight lines in Figure 4.10 reflect the fact that almost all the space in an R-tree index is used for leaf nodes, whose number varies linearly with the amount of data. For the linear-2 test configuration the total space occupied by the R-tree was about 40 bytes per data item, compared to 20 bytes per item for the index records alone. The corresponding figure for the quadratic-1/3 configuration was about 33

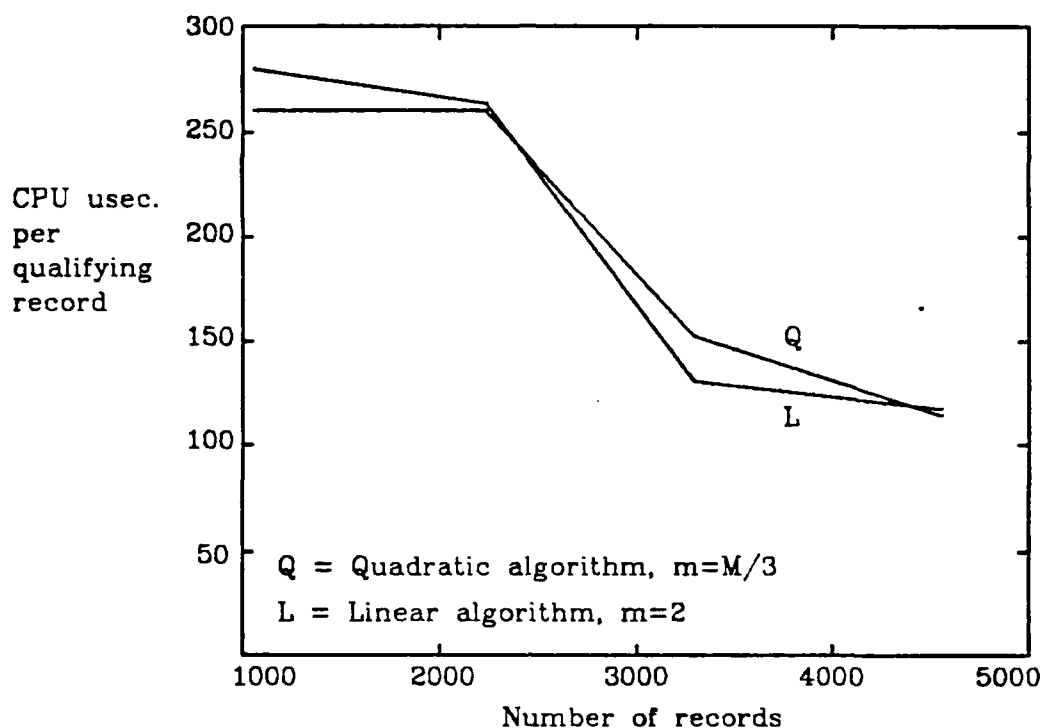


Figure 4.9
 Search performance vs. amount of data: CPU cost

bytes per item.

5. Conclusions

The R-tree structure has been shown to be useful for indexing spatial data objects that have non-zero size. Nodes corresponding to disk pages of reasonable size (e.g. 1024 bytes) have values of M that produce good performance. With smaller nodes the structure should also be effective as a main-memory index; CPU performance would be comparable but there would be no I/O cost.

The linear node-split algorithm proved to be as good as more expensive techniques. It was fast, and the slightly worse quality of the splits did not affect search performance noticeably.

Preliminary investigation indicates that R-trees would be easy to add to any relational database system that supported conventional access methods, (e.g. INGRES [9], System-R [1]). Moreover, the new structure would work especially well in conjunction with abstract data types and abstract indexes [14] to streamline the handling of spatial data.

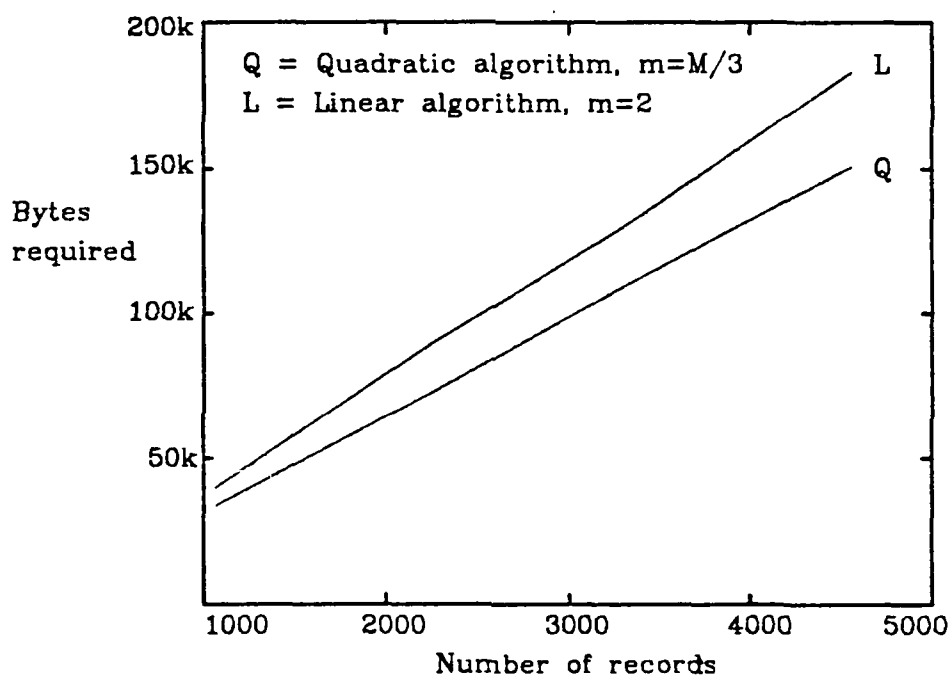


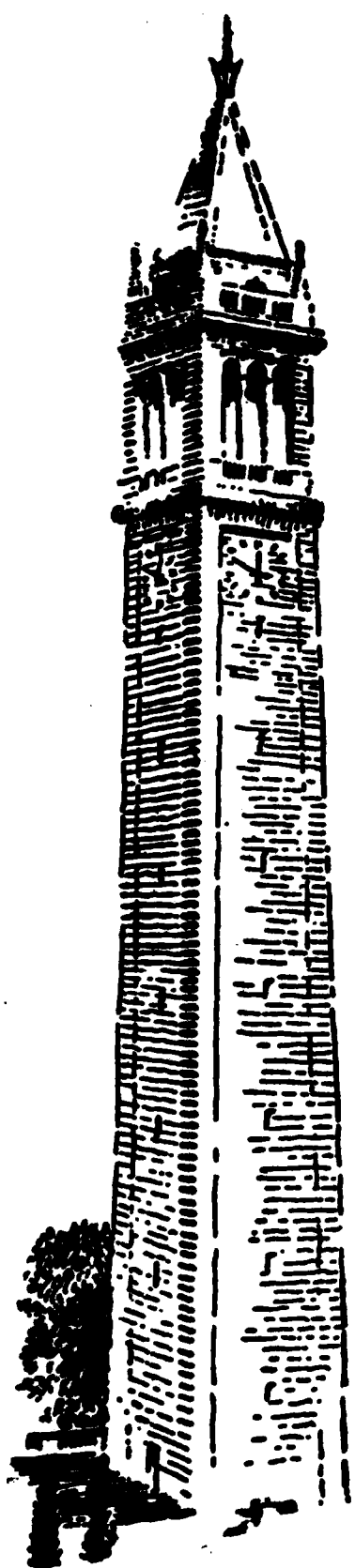
Figure 5.10
Space required for R-tree vs. amount of data.

REFERENCES

- [1] M. Astrahan, et al., "System R: Relational Approach to Database Management," *ACM Transactions on Database Systems* 1(2) pp. 97-137 (June 1976).
- [2] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indices," *Proc. 1970 ACM-SIGFIDET Workshop on Data Description and Access*, pp. 107-141 (Nov. 1970).
- [3] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM* 18(9) pp. 509-517 (September 1975).
- [4] J. L. Bentley, D. F. Stanat, and E. H. Williams Jr., "The complexity of fixed-radius near neighbor searching," *Inf. Proc. Lett.* 6(6) pp. 209-212 (December 1977).
- [5] J. L. Bentley and J. H. Friedman, "Data Structures for Range Searching," *Computing Surveys* 11(4) pp. 397-409 (December 1979).
- [6] D. Comer, "The Ubiquitous B-tree," *Computing Surveys* 11(2) pp. 121-138 (1979).
- [7] R. A. Finkel and J. L. Bentley, "Quad Trees - A Data Structure for Retrieval on Composite Keys," *Acta Informatica* 4 pp. 1-9 (1974).
- [8] A. Guttman and M. Stonebraker, "Using a Relational Database Management System for Computer Aided Design Data," *IEEE Database Engineering* 5(2)(June 1982).
- [9] G. Held, M. Stonebraker, and E. Wong, "INGRES - A Relational Data Base System," *Proc. AFIPS 1975 NCC* 44 pp. 409-416 (1975).
- [10] K. Hinrichs and J. Nievergelt, "The Grid File: A Data Structure Designed to Support Proximity Queries on Spatial Objects," Nr. 54, Institut fur Informatik, Eidgenossische Technische Hochschule, Zurich (July 1983).
- [11] M.G.H. Katevenis, R.W. Sherburne, D.A. Patterson, and C.H. Séquin, "The RISC II Micro-Architecture," *Proc. VLSI 83 Conference*, (August 1983).
- [12] J. K. Ousterhout, "Corner Stitching: A Data Structuring Technique for VLSI Layout Tools," Computer Science Report CSD 82/114, University of California, Berkeley (1982).
- [13] J. T. Robinson, "The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *ACM-SIGMOD Conference Proc.*, pp. 10-18 (April 1981).
- [14] M. Stonebraker, B. Rubenstein, and A. Guttman, "Application of Abstract Data Types and Abstract Indices to CAD Data Bases," Memorandum No. UCB/ERL M83/3, Electronics Research Laboratory, University of California, Berkeley (January 1983).
- [15] K. C. Wong and M. Edelberg, "Interval Hierarchies and Their Application to

Predicate Files," *ACM Transactions on Database Systems* 2(3) pp. 223-232 (September 1977).

- [16] G. Yuval, "Finding Near Neighbors in k-dimensional Space," *Inf. Proc. Lett.* 3(4) pp. 113-114 (March 1975).



QUEL AS A DATA TYPE

by

Michael Stonebraker, Erika Anderson, Eric Hanson
and Brad Rubenstein

Memorandum No. UCB/ERL M83/73

December 12, 1983

ELECTRONICS RESEARCH LABORATORY

College of Engineering 95 12 6 047

University of California, Berkeley, CA 94720

QUEL AS A DATA TYPE

by

Michael Stonebraker, Erika Anderson, Eric Hanson

and Brad Rubenstein

Memorandum No. UCB/ERL M83/73

December 12, 1983

ELECTRONICS RESEARCH LABORATORY

QUEL AS A DATA TYPE

by

Michael Stonebraker, Erika Anderson, Eric Hanson

and Brad Rubenstein

Memorandum No. UCB/ERL M83/73

December 12, 1983

ELECTRONICS RESEARCH LABORATORY

QUEL AS A DATA TYPE

by

Michael Stonebraker, Erika Anderson, Eric Hanson

and Brad Rubenstein

Memorandum No. UCB/ERL M83/73

December 12, 1983

ELECTRONICS RESEARCH LABORATORY

QUEL AS A DATA TYPE

by

Michael Stonebraker, Erika Anderson, Eric Hanson
and Brad Rubenstein

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA
BERKELEY, CA.

ABSTRACT

This paper explores the use of commands in a query language as an abstract data type (ADT) in data base management systems. Basically, an ADT facility allows new data types, such as polygons, lines, money, time, arrays of floating point numbers, bit vectors, etc., to supplement the built-in data types in a data base system. In this paper we demonstrate the power of adding a data type corresponding to commands in a query language. We also propose three extensions to the query language QUEL to enhance its power in this augmented environment.

I INTRODUCTION

Abstract data types (ADTs) [LISK74, GUTT77] have been extensively investigated in a programming language context. Basically, an ADT is an encapsulation of a data structure (so that its implementation details are not visible to an outside client procedure) along with a collection of related operations on this encapsulated structure. The canonical example of an ADT is a stack with related operations: new, push, pop and empty.

The use of ADTs in a relational data base context has been discussed in [ROWE79, SCHM78, WASS79]. In these proposals a relation is considered an abstract data type whose implementation details are hidden from application level software. Allowable operations are defined by procedures written in a programming language that supports both data base access and ADTs. One use

This Research was supported by the Navy Electronics Systems Command under Contract N00039-83-C-0243 and by the Air Force Office of Scientific Research under Grant 83-0021.

of this kind of data type is suggested in [ROWE79] and involves an EMPLOYEE abstract data type with related operations hire-employee, fire-employee and change-salary.

In [STON82, STON83] we presented an alternate use of ADTs. Instead of treating an entire relation as an ADT, we suggested that the individual columns of a relation be ADTs. This use of ADTs is a generalization of data base experts [STON80].

In Section II we briefly review our proposal and then in Section III we introduce QUEL as a data type and indicate desirable operators for this new type. Section IV turns to a discussion of three extensions to the QUEL language that are useful in this environment. In Section V we consider optimization issues related to QUEL ADTs. Lastly, we indicate that several data base problems including referential integrity, non-first normal form relations, and generalization hierarchies can be solved by defining QUEL as an abstract data type. Section VI presents our approach to these problems. Section VII concludes by summarizing the paper.

II ABSTRACT DATA TYPES

We explain our use of ADTs with an example concerning geometric objects. In computer aided design of integrated circuits, objects are often made up of rectangular boxes. For a VLSI data base one would like to define a column of a relation as type "box". For example, one might create a boxes relation as follows:

```
create boxes (owner = i4,  
              layer = c15,  
              box-desc = box-ADT)
```

Here, the boxes relation has three fields: the identifier of the circuit containing the box, the processing layer for the box (polysilicon, diffusion, etc.) and a description of the box's geometry. All fields are represented by built-in types

except box-desc which is an ADT.

Tuples can be appended to this relation using QUEL [STON76] as follows:

```
append to boxes (owner = 99,  
                 layer = "polysilicon",  
                 box-desc = "0,0,2,3")
```

The built-in data types are converted to an internal representation and stored in a data base system. The string "0,0,2,3", represents the box bounded by $x=0$, $y=0$, $x=2$, $y=3$ and requires special recognition code. An input procedure must be available to the DBMS to perform the conversion of the character string "0,0,2,3" to an object with data type box-ADT. Such a routine is analogous to the procedure `ascii-to-float` which converts a character string to a floating point number.

It is desirable to have special operators for box-ADTs. for example, one would clip box dimensions as follows:

```
range of b is boxes  
replace b (box-desc = b.box-desc * "0,0,4,1")  
       where b.owner = 99
```

The `*` operator represents box intersection. In this case "0,0,4,1" will be converted to an object of type box-ADT, and a procedure must be available to perform box intersection between this ADT and `b.box-desc`.

In addition, one might want to define new comparison operations. For example, one might wish to define `||` as an operator meaning "overlaps". The `||` operator could then be used to return the boxes overlapping the unit square based at the origin as follows:

```
range of b is boxes  
retrieve (b.box-desc)  
       where b.box-desc || "0,0,1,1"
```

Again, a procedure is required for the overlap operator.

As a result an ADT contains the following elements:

- a) a registration procedure to inform the DBMS of the new type, giving the length of its internal representation.
- b) a collection of routines which implement operators for this type and perform conversions to other types. These routines must obey a prespecified protocol for accepting arguments and returning results. Once defined by the ADT implementor, the new type and operators become available to other users of the DBMS.
- c) modest changes to the parser and query execution routines to correctly parse commands with new operators and call the routines defined by the ADT implementor during execution.

This abstraction has been constructed in about 2500 lines of code for the INGRES relational data base system. Implementation details are addressed in [FOGG82, ONG82], and ADTs execute with a modest performance degradation [FOGG82]. Suggestions concerning how to integrate new operators into query processing heuristics and access methods are contained in [STON83, ONG83].

III QUEL AS A DATA TYPE

We turn now to utilizing the ADT mechanism to define commands in a query language as an ADT. Hence, a column of a relation can have values which are one (or more) commands in the data manipulation language, QUEL. We explain our proposal using the following relations:

EMP (name, salary-history, hobbies, dept, age, bonus)
DEPT (dname, floor)
SALARY (name, date, pay-rate)

SOFTBALL (name, position, average)
MUSIC (name, instrument, level)
RACING (name, auto, circuit)

A tuple exists in the EMP relation for each employee in a particular company. Employees can have zero or more hobbies. For those employees who have softball as a hobby, a tuple in the SOFTBALL relation gives their position and batting average. If an employee plays an instrument, a tuple in MUSIC indicates the instrument he plays and his skill level. Lastly, those employees who race sportcars are listed in the RACING relation along with the type of car they drive and the circuit they race on.

The SALARY relation contains employees salary histories. Each time the salary of an employee is modified, a tuple is appended to the SALARY relation indicating the date of the modification and the new pay-rate. The DEPT relation contains the floor number of each department. Lastly, the EMP relation contains three fields, salary-history, hobbies, and dept which are of type QUEL. The hobbies field holds a query (or queries) which, when executed, will yield information on the employee's hobbies. The dept field contains a query which will return the name of the department for which the employee works, and the salary-history field contains a query that finds all records in his salary history. An example insert to the EMP relation might be:

```
append to EMP (
  name = "Fred",
  salary-history = "range of s is SALARY
                  retrieve (s.all)
                  where s.name = "Fred"",
  hobbies = "range of m is MUSIC
            retrieve (m.all) where m.name = "Fred"
            range of r is RACING
            retrieve (r.all) where r.name = "Fred"",
  dept = "range of d is DEPT
        retrieve (d.dname) where d.dname = "toy"",
  age = 25,
  bonus = 10)
```

The appropriate additional insertions are:

```
append to MUSIC(
  name = "Fred",
  instrument = "piano",
  level = "novice")
```

```

append to RACING(
    name = "Fred",
    auto = "formula Ford",
    circuit = "SCCA")

```

This collection of inserts will append Fred as a new employee in the toy department with racing and music as hobbies.

In a later section we will propose an implementation of this data type. In this section we specify desirable operators this type and their intended semantics.

The current implementation of ADTs [FOGG82, ONG82] allows operators to be overloaded. INGRES currently allows "." as an operator with two operands, a tuple variable and a column name, e.g E.name. Our first ADT operator overloads the operator ".". First, we propose that "." allow a left operand which is a field of type QUEL and a right operand of type column name. For example:

```

range of e is EMP
retrieve (e.hobbies.instrument)
    where e.name = "Fred"
    and e.hobbies.level = "novice"

```

In this case "name" is a column in the relation indicated by e while "level" and "instrument" are columns in the relation (or relations) specified by the QUEL in e.hobbies. This command is interpreted as follows:

- 1) Find all values for e.hobbies which satisfy the qualification "e.name = "Fred".
- 2) For each value found, ignore all commands which it contains except RETRIEVE and DEFINE VIEW. For each RETRIEVE command which the value contains, replace the keyword RETRIEVE with the keyword DEFINE VIEW and execute it to form a legal view. For each view definition which the value contains, execute it directly to form a legal view. Then, define t to be a tuple variable which will iterate over the this collection of views. For each one, execute:

```

retrieve (t.instrument) where t.level = "novice"

```

The result of the overall query is the union of the results of the individual commands executed in step 2.

In general, if X is a tuple variable, Y is a field of type QUEL, and Z is a field, then X.Y.Z is a field in a collection of views, one for each RETRIEVE and DEFINE

VIEW command contained in a qualifying value for X.Y. Moreover, "." can be arbitrarily nested and the above semantics apply recursively at each level. Also note that this use of "." is similar to that proposed in GEM [ZANI83], and we comment further on the relationship of our proposal to GEM in a later section.

Our second use of "." has a left operand which is a field of type QUEL and a right operand which is a QUEL statement, e.g.:

```
range of e is EMP
retrieve (e.salary-history.
    retrieve (date, pay-rate) where pay-rate < 400)
where e.name = "Fred"
```

Here, e.salary-history is a field of type QUEL and the inner RETRIEVE command is the right hand operand for the intervening ".". This use of "." is a short-hand notation for the equivalent expression:

```
range of e is EMP
retrieve (e.salary-history.date, e.salary-history.pay-rate)
where e.name = "Fred" and e.salary-history.pay-rate < 400
```

In this nested retrieval context "." has a similar meaning to the one discussed above. In particular, the left hand operator evaluates to the collection of views mentioned earlier, and a range variable, say t, is created to iteratively span this set. The QUEL command which is the right hand operand is then executed for each view by appending t as the tuple variable to any field name which does not have an explicit variable.

When the right hand operand is a RETRIEVE command, the result of this operator is a collection of result relations. The semantics of "." when the right hand operand is a QUEL update command are unclear, and we expect to support this form of nesting only for retrieves.

We now turn to several other operators on QUEL data items. First, all the normal character string operators can be overloaded. For example:

```
range of e is EMP
```



```

retrieve (e.name) where e.dept = "range of d is DEPT
                                retrieve (d.dname) where
                                d.dname = "toy""

```

In this context, "=" simply implies character string equality between e.dept and the constant string containing the query.

Consider an operator, ==, which has two fields of type QUEL as operands and returns true if they specify the same collection of tuples. For example,

```

range of e is EMP
range of f is EMP
retrieve (e.name, f.name) where e.salary-history ==
                                f.salary-history

```

This query will return pairs of employees with identical names and salary histories. A containment operator, <<, can be specified similarly for operands which are fields of type QUEL. Additionally, all operators in a relational algebra (e.g join, union, intersection) can be easily defined between fields of type QUEL.

Any relational algebra operators will produce a result of type relation. Since QUEL allows cascaded operators, we require operators for data of type relation. It is straight forward to overload all operators for the QUEL data type to apply to data of type relation. For example to find pairs of employees with different names and the same salary history, we would execute

```

range of e is EMP
range of f is EMP
retrieve (e.name, f.name)
    where e.salary-history.
           retrieve (date, payrate)
    == f.salary-history.
           retrieve (date, payrate)

```

Here, == has relations as both operands and returns true if the two relations are equal.

The last generalization is to allow any operator for fields of type QUEL to be overloaded to apply to operands which are QUEL statements or tuple variables. For example, suppose a relation STANDARD contains a collection of dates and

payrates. The following command would find all employees with the same salary history that appears in STANDARD:

```
range of e is EMP
range of s is STANDARD
retrieve (e.name)
  where e.salary-history.
        retrieve (date, payrate)
== retrieve (s.all)
```

Here the right hand operand of == is a simple QUEL statement. A shorthand for the above statement would have a tuple variable for the right operand of == as follows:

```
range of e is EMP
range of s is standard
retrieve (e.name)
  where e.salary-history.
        retrieve (date, payrate)
== s
```

Our complete set of proposed operators appears in Table 1. Most can be applied interchangeably to operands which are fields of type QUEL, tuple variables, QUEL statements, and relations.

IV EXTENSIONS TO QUEL

There are three main extensions which we propose for inclusion in QUEL to enhance its power in the ADT environment of Section III. In addition, we endorse the proposal made in [ZANI83] to have default tuple variables. In this situation, a command such as:

```
retrieve (EMP.age) where EMP.name = "Fred"
```

would be interpreted as:

```
range of EMP is EMP
retrieve (EMP.age) where EMP.name = "Fred"
```

This suggestion simplifies many QUEL commands and was inserted into one version of QUEL [RT183].

operator name	description	left operand	right operand	result
.	referencing	field of type QUEL	field-name	field
.	referencing	field of type QUEL	QUEL statement	relation
=	character string compare	*	*	boolean
==	relation compare	*	*	boolean
>>	relation inclusion	*	*	boolean
<<	relation inclusion	*	*	boolean
U	union	*	*	relation
!!	intersection	*	*	relation
JJ	natural join	*	*	relation
OJ	outer join	*	*	relation

* denotes a field of type QUEL, a QUEL statement, a relation or a tuple variable

Proposed Operators

Table 1

In addition to default tuple variables we propose three other extensions. First, we suggest the possibility of executing data in the data base rather than retrieving or updating it. The syntax is as follows:

`exec (EMP.hobbies) where EMP.name = "Fred"`

The target list must be a field of type QUEL and instances which satisfy the qualification are found and executed. In this case, the hobbies which Fred engages in are returned.

This extension frees a user from having to know the field names in the QUEL in e.hobbies. Also, it allows one to store updates in the data base and execute them at a later time. Such data base procedures are discussed in Section VI.

Notice that EXEC complicates the extended interpretation of "." in the previous section. For example, it is reasonable to have a value for e.hobbies which is an EXEC command. For example, one could change Fred's hobbies to be the same as John's by the following update:

```
range of E is EMP
replace e (hobbies =
    "range of f is EMP
    exec (f.hobbies) where f.name = "John""
where e.name = "Fred"
```

If X is a tuple variable, Y is a field of type QUEL and Z is a field and if a qualifying value for X.Y contains an EXEC command, then the semantics of X.Y.Z from the previous section must be extended. In particular X.Y.Z can be a column in an additional set of views. For each EXEC contained in a qualifying value of X.Y, replace the EXEC by RETRIEVE and run the command. If the result contains values of type QUEL, then X.Y.Z must span any views which result from these values by executing DEFINE VIEW commands, replacing RETRIEVE commands by DEFINE VIEW commands and recursively applying the above meaning to EXEC commands.

The second extension is to generalize the range statement. We propose to allow a tuple variable to range over a collection of one or more relations. Then we use this facility to support the further generalization illustrated below:

```
range of e is
    EMP.salary-history where EMP.name = "Fred"
retrieve (e.date) where e.pay-rate = 1000
```

The intent is to allow e to range over the result of a query specification. Because RETRIEVE is the only reasonable QUEL command to put in a range statement, we leave it out of the syntax and include only the target list and qualification.

Moreover, the query specification must return data items of type QUEL. The purpose of the second extension is to allow the above expression rather than the less natural equivalent command:

```
range of e is EMP
retrieve (e.salary-history.date)
  where e.salary-history.pay-rate = 1000
  and e.name = "Fred"
```

If X and U are tuple variables and Y a field of type QUEL, then the semantics of

range of U is X.Y where qualification

are the following:

- 1) Run the query
retrieve (X.Y) where qualification
to find qualifying data items of type QUEL.
- 2) For each RETRIEVE, DEFINE VIEW or EXEC command, perform the steps indicated earlier to define the appropriate collection of views, C1,...,Cn.
- 3) Replace the range statement by
range of U is C1,...,Cn

The third extension is to allow update commands to have a generalized target relation as suggested by the following example:

```
append to EMP.salary-history
  (date = "6/81", payrate = 2000, name = "Fred")
where EMP.name = "Fred"
```

Currently QUEL only supports a target which is a relation. In this generalization, the target can also be a column of a relation in the data base which is of type QUEL.

The intent of the third extension is to allow the above expression rather than the equivalent extended command:

```
range of e is EMP.salary-history where EMP.name = "Fred"
append to e (date = "6/81", payrate = 2000, name = "Fred")
```

Notice that extended range statements and extended targets automatically introduce views. The usual semantic problems occur in updating these views.

V SPECIAL CASES OF QUEL AS A DATA TYPE

Three special QUEL data types will be suggested in this section to allow either increased performance or a more natural syntax. First we suggest relations as a special case of the QUEL ADT. Clearly, a value of type QUEL can be a relation, i.e.:

range of R is any-relation
retrieve (R.all)

Since the interpretation of the QUEL extensions in Section IV required that the query be treated as a view, we must invoke view processing to support such functions. A data type of relation as a special case of a QUEL data type will allow such operators to be optimized by ignoring the view processing.

The internal representation of a QUEL data type may be anything from a text string for the command to a machine language procedure containing a compiled version of the access plan. The choice depends on trading off efficiency, flexibility and complexity of the underlying DBMS. Alternatively, it is also possible to precompute the answer to any RETRIEVE command. This collection of pointers to tuples would be stored as the value of the field. In the case that at most one tuple qualified, this value would be a pointer to a single tuple or the null pointer. This representation is exactly the data type "pointer to a tuple" suggested by Powell [POWE83] and by Zaniola [ZANI83]. More generally, the value could contain multiple pointers to tuples in different relations. Consequently, implementing the QUEL data type by precomputing answers for QUEL queries provides a generalized version of previous proposals. Storing such physical pointers in the data base has a clear speed advantages over storing the query. However, it also has the disadvantage that a pointer can be left "dangling" if the tuple it points to is moved. Moreover, no consistency guarantee is made if the tuple which is pointed to gets updated. Hence, precomputing answers should be considered a very dangerous way to obtain

efficiency.

The third special case of a QUEL data type can be illustrated by appending a tuple to the EMP relation, e.g.

```
append to EMP (name = "Joe", dept = "shoe")
```

In this case dept is a field of type QUEL and we would prefer to simply enter the value "shoe" and not the remainder of the query. If dept is defined to be a new ADT which is special version of the QUEL ADT, then the routine which converts from external to internal format for this ADT can change "shoe" to:

```
retrieve (DEPT.dname) where DEPT.dname = "shoe"
```

Consequently, a user need not type all the extra pieces of the QUEL command.

VI USES OF QUEL AS A DATA TYPE

In this section we indicate several uses for the above facilities.

6.1 Unnormalized Relation

There has been much discussion surrounding normalization of relations, and several recent proposals have advocated unnormalized relations [HASK82, GUTT82, ZANI83]. One use of a QUEL ADT is to support hierarchical data as noted in the example use of salary-history.

6.2 Referential Integrity

The notion of referential integrity has been formalized for relational data bases in [DATE81]. Basically, a data item must take on values from the set of values in a column of another relation. Notice that our example use of the dept field in the EMP relation automatically has this property. Although not all of the options suggested in [DATE81] can be easily supported using QUEL as a data type, several of the more common ones can be.

6.3 Variant Records

Our use of queries in the hobbies field corresponds closely to the notion of variant records in a programming language such as Pascal. Frames oriented languages such as FRL [ROBE77] or KRL [BOBR77] also allow a slot in a frame to contain a value of an arbitrary type with arbitrary fields. Our use of QUEL queries with different ranges supports this notion.

6.4 Aggregation and Generalization

QUEL as a data type can support both generalization and aggregation as proposed in [SMIT77]. For example, consider:

```
PEOPLE (name, phone#)
```

where phone# is of type QUEL and is an aggregate for the more detailed values area-code, exchange and number. A simple append to PEOPLE might be:

```
append to PEOPLE (name = "Fred", phone# =  
    "retrieve (area-code = 415,  
        exchange = 999,  
        number = 9911)")
```

Generalization is also easy to support. If all employees have exactly one hobby, then the hobbies field in the EMP example relation will specify a simple generalization hierarchy. In fact, our example use of hobbies supports a generalization hierarchy with members which can be in several of the subcategories at once.

6.5 Data Base Procedures

Stored commands are easily supported with the facilities described above. For example, suppose an employee is allowed to have only one hobby and we want a general data base procedure to change the hobby of an employee from playing softball to playing a musical instrument. Call this procedure "softball-to-music" and add it to a relation PROCEDURES as follows:


```

append to PROCEDURES(
name = "softball-to-music",
code = "delete SOFTBALL where SOFTBALL.name = $1
      append to MUSIC (name = $1,
                      instrument = $2,
                      level = $3)
      replace EMP (hobbies =
                  "retrieve (MUSIC.all)
                  where MUSIC.name = $1")

```

Now suppose we define a new ADT operator, WITH, that will substitute a parameter list given as the right hand operator into a query which is the left hand operator. With this operator we can make Fred play the violin at skill level novice as follows:

```

exec (PROCEDURES.code WITH (Fred, violin, novice)) where
PROCEDURES.name = "softball-to-music"

```

In this way we can store collections of QUEL commands in the data base and execute them as procedures.

6.6 Triggers

Triggers have been widely suggested as a possible mechanism for implementing consistency constraints and for producing side effects for commands. They can be supported by using the features discussed in previous sections. Consider a relation:

```

TRIGGER (if, relname, command, then)

```

The field "then" is of type QUEL while "if" is of type QUEL qualification. Both "relname" and "command" are ordinary character string fields.

Currently INGRES performs deferred update [STON76] and writes a "side file" containing proposed changes to the data base as phase 1 of a command. In phase 2 the side file is processed and the changes are installed. Consider modifying the side file to be a relation SIDE and interrupting query processing at the end of phase 1 to perform:

```

exec (TRIGGER.then) where TRIGGER.if
    and TRIGGER.command = user-command
    and TRIGGER.relname = relation-from-user

```

Here, user-command is the type of command run by the user (e.g. replace, delete) and relation-from-user is the name of the relation being updated. These constants are readily available from the run time DBMS.

An example tuple in the TRIGGER relation might be:

```

append to TRIGGER(
    if = "SIDE.TID = EMP.TID and EMP.name = "Fred"
        and SIDE.age > EMP.age",
    relname = "EMP",
    command = "replace",
    then = "append to ALARM
        (message = "Fred got older")")

```

The TRIGGER relation is used to provide an alerting capability when Fred receives an update. Since TRIGGER may have a large collection of tuples, we require indexing on relname and command to restrict the set of TRIGGER.if terms that must be evaluated. It may be reasonable to have other extra fields in TRIGGER to provide further efficiency in TRIGGER selection.

6.7 Storing Data as Rules

Consider the requirement that all employees over 40 years old must receive a bonus of 1000. The relation in Section II showed both "age" and "bonus" as explicit data and an integrity constraint could easily be defined to enforce this constraint, e.g.:

```

range of e is EMP
define integrity E where E.bonus = 1000 or E.age <= 40

```

However, an alternative representation would be to remove "bonus" as a stored field in EMP and add the following rule to TRIGGER:

```

append to TRIGGER(
    relname = "EMP"
    then = "replace SIDE( bonus = 1000)
        where SIDE.TID = EMP.TID and

```

EMP.age > 40"

If the QUEL parser was changed to allow queries that retrieve fields which are not stored, then this trigger will return the correct data by updating SIDE. Hence, the trigger mechanism can support storing data items as rules. Of course, the efficiency of this implementation is questionable, and it is awkward to ask questions about what rules are in effect.

6.8 Complex Objects

There has been substantial discussion concerning data base support for complex objects [LOR83, STON83]. Suppose a complex object is composed of text, lines, and polygons. It would be possible to construct the following relations:

OBJECT (Oid, description)
LINE (Lid, description, location)
TEXT (Tid, description, location)
POLYGON (Pid, description, location)

Here, the LINE, TEXT and POLYGON relations hold descriptions of individual objects and can make use of the abstract data types described in [STON83]. Then, the description field in OBJECT would be of type QUEL and contain queries to assemble the pieces of any given object from the other relations. This representation allows clean sharing of Lines, Text and Polygons among multiple higher level objects by allowing the same query to appear in multiple object descriptions.

Materializing an object from the OBJECT relation will be slow since it involves executing several additional QUEL queries. Hence, it may be desirable to precompute the value of frequently used objects and store the result in the OBJECT description field. This has the same costs and benefits which were discussed in the context of storing tuple identifiers instead of queries in the preceding section.

6.9 Transitive Closure

The facilities of this paper can be used to support transitive closure operations such as found in the "parts explosion" problem. Suppose one creates a PARTS relation as follows:

```
PARTS (pname, composed-of)
```

Consider a car which is made up of a drivetrain and a body. These are made up in turn of other smaller parts. The car would be inserted as follows:

```
append to PARTS(  
  pname = "car",  
  composed-of = "retrieve (pname = "car")  
                exec (PARTS.composed-of  
                    where PARTS.pname = "drive-train"  
                exec (PARTS.composed-of  
                    where PARTS.pname = "body")")
```

The command

```
exec (PARTS.composed-of) where PARTS.pname = "car"
```

will find all the parts that make up a car.

VII IMPLEMENTATION

If INGRES had been designed to support internal multitasking, then it would be a simple matter to implement EXEC by stacking the INGRES processing environment and executing the new command in a single INGRES process. However, at this point it would be very costly to change our code to be reentrant and support this kind of recursion. Other systems (e.g. System-R [ASTR76]) do not have this shortcoming.

Hence, our operational code to implement EXEC spawns a separate copy of the INGRES code and passes the QUEL command to the spawned version for execution. Returned data is redirected through the INGRES which did the spawning to the user who ran the original command. Since the passed command can be another EXEC, the total number of spawned INGRES's can increase

without bound. Currently, the command is passed to the spawned process as a character string and all query processing steps are performed at run time by the second process.

We are currently implementing QUEL as an ADT. This data type is internally represented as a character string. Storing a preprocessed version of the command would entail a great deal more code. Operators which return a result of type relation will store the result in the data base and return the name of the object. This result can be involved in further processing or returned to the user. In the latter case, it is the responsibility of the internal-to-external conversion routine to accept the relation name, access the data base and return tuples to the calling program or user.

No thought has been given on how to optimize QUEL commands extended with the operators of Table 1. Integrating these new functions into query processing heuristics is left for future research. The design of a programming language interface supporting the objects generated by our proposal also remains to be studied.

VIII CONCLUSIONS

This paper has proposed a novel use of abstract data types and extended QUEL with three additional features. These extensions support added power, referential integrity, variant records, data base procedures, generalization and aggregation in a single facility.

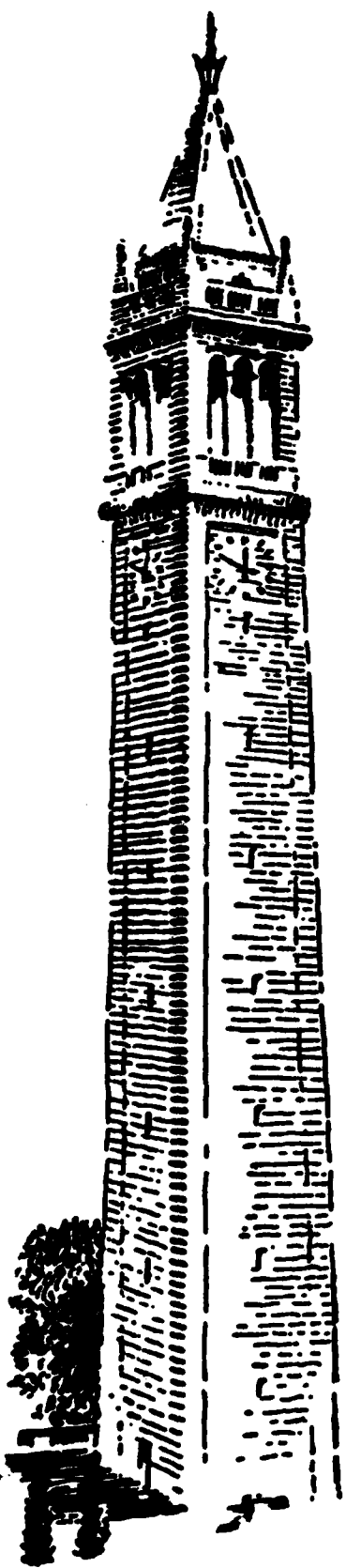
Our proposal has points in common with GEM which supports new data types corresponding to "pointer to a tuple" and "set of values". Moreover, generalization hierarchies are supported and range variables can conveniently be defined over entities in this hierarchy. Our proposal effectively supports both of GEM's new data types as special cases of the QUEL ADT. Moreover, generalization is cleanly supported. Only GEM's use of range variables is not

contained in our proposal.

REFERENCES

- [ASTR76] Astrahan, M. et. al., "System R: A Relational Approach to Data," ACM TODS, June 1976.
- [BOBR77] Bobrow, D. and Winograd, T., "An Overview of KRL, a Knowledge Representation Language," Cognitive Science, 1,1 1977
- [DATE81] Date, C., "Referential Integrity," Proc. 6th VLDB Conference, Cannes, France, September 1981.
- [FOGG82] Fogg, D., "Implementation of Domain Abstraction in the Relational Database System, INGRES", Masters Report, EECS Dept, University of California, Berkeley, Ca Sept. 1982.
- [GUTT77] Gutttag, J., "Abstract Data Types and the Development of Data Structures," CACM, June 1977.
- [GUTT82] Guttman, A. and Stonebraker, M., "Using a Relational Database Management System for Computer Aided Design Data", Data Base Engineering, June 1982.
- [HASK82] Haskins, R. and Lorie, R., "On Extending the Functions of a Relational Database System," Proc. 1982 ACM-SIGMOD Conference on Management of Data, Orlando, Fl, June 1982.
- [LORI83] Lorie, R. and Plouffe, W., "Complex Objects and Their Use in Design Transactions," Proc. Engineering Design Applications of ACM-IEEE Data Base Week, San Jose, Ca., May 1983.
- [LSK74] Liskov, B. and Zilles, S., "Programming With Abstract Data Types," ACM-SIGPLAN Notices, April 1974.
- [ONG82] Ong, J., "The Design and Implementation of Abstract Data Types in the Relational Database System, INGRES," Masters Report, EECS Dept, University of California, Berkeley, Ca Sept. 1980.
- [ONG83] Ong, J., et. al., "Implementation of Data Abstraction in the Relational Database System INGRES," to appear in SIGMOD Record.
- [POWE83] Powell, M. and Linton, M., "Database Support for Programming Environments," Proc. Engineering Design Applications of ACM-IEEE Database Week, San Jose, Ca., May 1983.
- [RTI83] "INGRES Reference Manual, Version 1.4," Relational Technology, Inc., Berkeley, Ca., 1983.
- [ROBE77] Roberts, R. and Goldstein, I., "The FRL Manual," MIT, AI Laboratory, Memo No. 409, Sept 1977.
- [ROWE79] Rowe, L. and Schoens, K., "Data Abstraction, Views and Updates in RIGEL," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass. May 1979.
- [SCHM78] Schmidt, J., "Type Concepts for Database Definition," Proc. International Conference on Data Bases, Haifa, Israel, August 1978.
- [SMIT77] Smith, J and Smith, D., "Database Abstractions: Aggregation and Generalization," ACM TODS, June 1977.

- [STON76] Stonebraker, M. et al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976.
- [STON80] Stonebraker, M., and Keller, K., "Embedding Expert Knowledge and Hypothetical Data Bases in a Data Base System," Proc 1980 ACM-SIGMOD Conference on Management of Data, Santa Monica, Ca., May 1980.
- [STON82] Stonebraker, M., "Adding Semantic Knowledge to a Relational Database System," Proc. NSF Workshop on Semantic Modeling, Intervale, N.H. June 1982 (to appear as Springer-Verlag book edited by M. Brodie).
- [STON83] Stonebraker, M., et. al. "Application of Abstract Data Types and Abstract Indices to CAD Databases," Proc. Engineering Design Applications of ACM-IEEE Database Week, San Jose, Ca., May 1983.
- [WASS79] Wasserman, A.I., "The Data Management Facilities of PLAIN," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., May 1979.
- [ZANI83] Zaniola, C., "The Database Language GEM," Proc. 1983 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., May 1983.



Implementation Techniques
for
Main Memory Database Systems

by

David J. DeWitt, Randy H. Katz,
Frank Olken, Leonard D. Shapiro,
Michael R. Stonebraker and David Wood

Memorandum No. UCB/ERL 84/5

23 January 1984

ELECTRONICS RESEARCH LABORATORY
College of Engineering 85 12 6 047
University of California, Berkeley, CA 94720

Implementation Techniques
for
Main Memory Database Systems

David J. DeWitt¹
Randy H. Katz²
Frank Olken³
Leonard D. Shapiro⁴
Michael R. Stonebraker²
David Wood²

¹ Computer Sciences Department, University of Wisconsin

² EECS Department, University of California at Berkeley

³ CSAM Department, Lawrence Berkeley Laboratory

⁴ Department of Computer Science, North Dakota State University

This research was partially supported by the National Science Foundation under grants MCS82-01860, MCS82-01870, by the Department of Energy under contracts #DE-AC02-81ER10920 and #W-7405-ENG-48, and by the Air Force Office of Scientific Research under Grant S3-0021.

ABSTRACT

With the availability of very large, relatively inexpensive main memories, it is becoming possible to keep large databases resident in main memory. In this paper we consider the changes necessary to permit a relational database system to take advantage of large amounts of main memory. We evaluate AVL vs. B+ -tree access methods for main memory databases, hash-based query processing strategies vs. sort-merge, and study recovery issues when most or all of the database fits in main memory. As expected, B+ -trees are the preferred storage mechanism unless more than 80-90% of the database fits in main memory. A somewhat surprising result is that hash based query processing strategies are advantageous for large memory situations.

Key Words and Phrases: Main Memory Databases, Access Methods, Join Algorithms, Access Planning, Recovery Mechanisms

1. Introduction

Throughout the past decade main memory prices have plummeted and are expected to continue to do so. At the present time, memory for super-minicomputers such as the VAX 11/780 costs approximately \$1,500 a megabyte. By 1990, 1 megabit memory chips will be commonplace and should further reduce prices by another order of magnitude. Thus, in 1990 a gigabyte of memory should cost less than \$200,000. If 4 megabit memory chips are available, the price might be as low as \$50,000.

With the availability of larger amounts of main memory, it becomes possible to contemplate the storage of databases as main memory objects. In fact, IMS Fast Path [DATE82] has supported such databases for some time. In this paper we consider the changes that might be needed to a relational database system if most (or all) of a relation(s) is (are) resident in main memory.

In Section 2, the performance of alternative access methods for main memory database systems are considered. Algorithms for relational database operators in this environment are presented and evaluated in Section 3. In Section 4, we describe how access planning will be affected by the availability of large amounts of main memory for query processing. Section 5 discusses recovery in memory resident databases. Our conclusions and suggestions for future research are contained in Section 6.

2. Access Methods for Memory Resident Databases

The standard access method for data on disk is the B+-tree [COME79], providing both random and sequential key access. A B+-tree is specially designed to provide fast access to disk-resident data and makes fundamental use of the page size of the device. On the other hand, if a keyed relation is known to reside in main memory, then an AVL (or other binary) tree organization may be a better choice. In this section we analyze the performance of both structure for a relation R with the following characteristics:

$ R $	number of tuples in relation R
K	width of the key for R in bytes
L	width of a tuple in bytes
P	page size in bytes
4	size of a pointer in bytes

We have analyzed two cases of interest. The first is the cost of retrieving a single tuple using a random key value. An example of this type of query is:

retrieve (emp.salary) where emp.name = "Jones"

The second case analyzed is the cost of reading N records sequentially. Consider the query

retrieve (emp.salary, emp.name) where emp.name = "J*"

which requests data on all employees whose names begin with J. To execute this query, the database system would locate the first employee with a name beginning with J and then read sequentially. This second case analyzes the sequential access portion of such a command.

For both cases (random and sequential access), there are two costs that are specific to the access method:

- |page reads| the number of pages read to execute the query
- |comparisons| the number of record comparisons required to isolate the particular data of interest.

The number of comparisons is indicative of the CPU time required to process the command while the number of page reads approximates the I/O costs.

To compare the performance AVL and B+-trees, we propose the following cost function:

$$\text{cost} = Z * |\text{page-reads}| + |\text{comparisons}|$$

Since a page read consumes perhaps 2000 instructions of operating system overhead and 30 milliseconds of elapsed time while a comparison can easily be done in 200, we expect realistic values of Z to be in the range of 10 to 30. Later in the section we will use several values in this range.

Moreover, it is possible (although not very likely) that an AVL-tree comparison will be cheaper than a B+-tree comparison. The reasoning is that the B+-tree record must be located within a page while an AVL tree does not contain any page structure and records can be directly located. Consequently, we assume that an AVL-tree comparison costs Y times a B+-tree comparison for some $Y < 1$.

From Knuth [KNUT73], we can observe that in an $|R|$ -tuple AVL tree approximately

$$C = \log_2 |R| + 0.25 \text{ comparisons}$$

are required to find a tuple in a relation. Without any special precautions each of the C nodes to be inspected will be on a different page.¹ Hence, the number of pages accessed is approximately C . The

¹ If a paged binary tree organization is used instead, the fanout per node will be slightly worse than the B-tree. Furthermore,

AVL structure will occupy approximately

$$S = \left\lceil \frac{||R|| * (L + 8)}{P} \right\rceil \text{ pages}$$

Here $\lceil X \rceil$ denotes the smallest integer larger than X . If $|M|$ pages of main memory are available, and if $|M| < |S|$, and if a random replacement algorithm is used, the number of page faults to find a tuple in a relation will be approximately:

$$faults = C * (1 - \frac{|M|}{S})$$

Consequently the cost of a random access by key is:

$$cost(AVL) = Z * C * (1 - \frac{|M|}{S}) + Y * C$$

Next we derive the approximate cost for a random access to a tuple using a B+ -tree. According to YAO [YAO78], B-tree nodes are approximately 69 percent full on the average. Hence, the fanout of a B+ -tree is approximately

$$A = \frac{.69 * P}{K + 4}$$

The number of leaf nodes will be about

$$D = \frac{||R|| * L}{.69 * P} \text{ data pages}$$

The height of a B+ -tree index is thereby

$$height = \left\lceil \frac{\log_2 D}{\log_2 A} \right\rceil$$

The number of comparisons required to locate a tuple with a particular value is:

$$C' = \lceil \log_2 ||R|| \rceil$$

The number of pages which the tree consumes is about

$$S' = D + \left\lceil \frac{D}{A} \right\rceil + \left\lceil \frac{1}{A} \right\rceil \left\lceil \frac{D}{A} \right\rceil + \left\lceil \frac{1}{A} \right\rceil \left\lceil \frac{1}{A} \right\rceil \left\lceil \frac{D}{A} \right\rceil + \dots = D \sum_{i=0}^{\infty} \left\lceil \frac{1}{A} \right\rceil^i$$

To a first approximation S' is

$$S' = D * \frac{A}{A-1}$$

Again the number of page faults is approximately

paged binary trees are not balanced and the worst case access time may be significantly poorer than in the case of a B-tree.

$$faults = (height + 1) * (1 - \frac{|M|}{S'})$$

As a result the cost of a B+ -tree access by key is:

$$cost(B+ -tree) = Z * (height + 1) * (1 - \frac{|M|}{S'}) + C'$$

An AVL-Tree will be the preferred structure for case 1 if

$$DIFF = cost(B+ -tree) - cost(AVL-Tree) > 0$$

If we assume that $C = C' = \log_2 ||R||$ and rearrange the terms in the inequality, then an AVL-Tree will be preferred if:

$$(1 - Y) * \log_2 ||R|| > Z * \log_2 ||R|| * (1 - \frac{|M|}{S'}) - Z * (height + 1) * (1 - \frac{|M|}{S'})$$

Note that if $L \gg 8$ then $S \simeq 0.69 * S'$. Define $H = \frac{height + 1}{\log_2 ||R||}$. Some simplification yields:

$$\frac{|M|}{S} > \frac{Z * (1 - H) + Y - 1}{Z} * \frac{1 - H}{1.45}$$

Obviously, if $|M| > S$, then AVL trees are the preferred structure regardless of the values of H, Y, and Z.

In this situation, the entire AVL-Tree is resident in main memory and there are no disk accesses. Since both data structures require the same number of comparisons and the AVL comparisons are cheaper, then the AVL-Tree is guaranteed to have lower cost. If $|M| < S$ then AVL trees will be preferred if the value of $\frac{|M|}{S}$ is larger than the value of $\min(|M|/S)$ shown as in Table 1. As can be seen, essentially all

of a relation has to be resident in main memory before an AVL tree is the preferred structure. For rea-

Table 1 - Minimum Residency Factor For Random Access

Z	Y	H	min (M /S)
10	.5	.1	.91
10	.5	.2	.87
10	.5	.3	.82
10	.75	.1	.94
10	.75	.2	.90
10	.75	.3	.86
15	.75	.1	.96
15	.75	.2	.91
15	.75	.3	.86

sonable values of H , Y and Z , at least 80 percent and sometimes more than 90 percent of a relation must be main memory resident.

We turn now to sequential access. For an AVL-Tree, the cost of reading N records sequentially is N comparisons and N page reads, i.e.:

$$\text{seq-cost}(\text{AVL}) = Y*N + N*Z*(1 - \frac{|M|}{S})$$

On the other hand, N records in a B+ -Tree will occupy

$$Q = \left\lceil \frac{(N*L)/(1.38*P)}{L/(.69*P)} \right\rceil \text{ data pages}$$

and consequently:

$$\text{seq-cost}(\text{B+ -Tree}) = N + Q*Z*(1 - \frac{|M|}{S'})$$

An AVL-Tree will be preferred if:

$$\frac{|M|}{S} > \frac{Z(1-H') + (Y-1)}{Z*(1-H'/1.45)}$$

where $H' = \frac{Q}{N}$. It appears that reasonable values for H' are similar to reasonable values for H ; hence,

Table 1 also applies to sequential access.

In both random and sequential access, a very high percentage of the tree must be in main memory for an AVL-Tree to be competitive. Hence, it is likely to be a structure of limited general utility and B+ -Trees will continue to remain the dominant access method for database management systems.

3. Algorithms for Relational Database Operations

3.1. Introduction

In this section we explore the performance of alternative algorithms for relational database operations in an environment with very large amounts of main memory. Since many of the techniques used for executing the relational join operator can also be used for other relational operators (e.g. aggregate functions, cross product, and division), our evaluation efforts have concentrated on the join operation. However, at the end of the section, we discuss how our results extend to these other algorithms.

After introducing the notation used in our analysis, we present an analysis of the familiar sort-merge [BLAS77] join algorithm using this notation. Next we analyze a multipass extension of the

simple hashing algorithm. The third algorithm described is an algorithm that has been proposed by the Japanese 5th generation project [KITS83], and is called GRACE. In the first phase, the join of two large relations is reduced to the join of several small sets of tuples. During the second phase, the tuple sets are joined using a hardware sorter and a sort-merge algorithm. Finally, we present a new algorithm, called the Hybrid algorithm. This algorithm is similar to the GRACE algorithm in that it partitions a join into a set of smaller joins. However, during the second phase, hashing is used instead of sort merge.

In the following sections we develop cost formulas for each of the four algorithms and report the result of analytic simulations of the four algorithms. Our results indicate that the Hybrid algorithm is preferable to all others over a large range of parameter values.

3.2. Notation and Assumptions

Let R and S be the two relations to be joined. The number of pages in these two relations is denoted $|R|$ and $|S|$, respectively. The number of tuples in R and S are represented by $||R||$ and $||S||$. The number of pages of main memory available to perform the join operation is denoted as $|M|$. Given $|M|$ pages of main memory, $\{M\}_R$, $\{M\}_S$ specify the number of tuples from R and S that can fit in main memory at one time.

We have used the following parameters to characterize the performance of the computer system used:

comp	time to compare keys
hash	time to hash a key
move	time to move a tuple
swap	time to swap two tuples
IO_{SEQ}	time to perform a sequential IO operation
IO_{RAND}	time to execute a random IO operation

To simplify our analysis we have made a number of assumptions. First, we have assumed that $|R| \leq |S|$. Next, several quantities need to be incremented by slight amounts to be accurate. For example, a hash table or a sort structure to hold R requires somewhat more pages than $|R|$, and finding a key value in a hash table requires, on the average, somewhat more than one probe. We use "F" to denote any and all of these increments, so for example a hash table to hold R will require $|R|*F$ pages. To simplify cost calculations, we have assumed no overlap of CPU and IO processing. We have also ignored the

cost of reading the relations initially and the cost of writing the result of the join to disk since these costs are the same for each algorithm.

In any sorting or hashing algorithm, the implementor must make a decision as to whether the sort structure or hash table will contain entire tuples or only Tuple IDs (TIDs) and perhaps keys. If only TIDs or TID-key pairs are used, there is a significant space savings since fewer bytes need to be manipulated. On the other hand, every time a pair of joined tuples is output, the original tuples must be retrieved. Since these tuples will most likely reside on disk, the cost of the random accesses to retrieve the tuples can exceed the savings of using TIDs if the join produces a large number of tuples. Fortunately, we can avoid making a choice as the decision affects our algorithms only in the values assigned to certain parameters. For example, if only TID-key pairs are used then the parameter measuring the time for a move will be smaller than if entire tuples are manipulated.

Three algorithms (Sort-merge, GRACE, and Hybrid hash) are much easier to describe if they require at most two passes. Hence we assume the necessary condition $\sqrt{|S| * F} \leq |M|$. For example, if $F = 1.2$, then $|M|$ is only 1,000 pages (4 megabytes at 4K bytes/page), and $|S|$ (and $|R|$, since $|R| \leq |S|$) can be as large as 800,000 pages (3.2 gigabytes)!

3.3. Partitioning a Relation by Hash Values

If $|M| < |R| * F$, each of the hashing algorithms described in this paper requires that R and/or S be partitioned into distinct subsets such that any two tuples which hash to the same value lie in the same subset. One such partitioning is into the sets R_x such that R_x contains those tuples r for which $h(r) = x$. We call such a partition *compatible* with h .

A general way to create a partition of R compatible with h is to partition the set of hash values X that h can assume into subsets, say X_1, \dots, X_n . Then, for $i = 1, \dots, n$ define R_i to be all tuples r such that $h(r)$ is in X_i . In fact, every partition of R compatible with h can be derived in this general way, beginning with a partition of the hash values. The power of this method is that if we partition both R and S using the same h and the same partition of hash values, say into R_1, \dots, R_n and S_1, \dots, S_n , then the problem of joining R and S is reduced to the task of joining R_1 with S_1 , R_2 with S_2 , etc. [BABB79, GOOD81].

In order for the hash table of each set of R tuples to fit in memory, $|R_i| * F$ must be $\leq |M|$. This is not easily guaranteed. For example, how can one choose a partition of R , compatible with h , into two sets of equal size? One might try trial and error: Begin by partitioning the set of hash values into two sets X_1 and X_2 of equal size and then consider the sizes of the two corresponding sets of tuples R_1 and R_2 . If the R -sets are not of equal size then one changes the X sets to compensate, check the new R -sets again, etc. Despite the apparent difficulties of selecting the sets X_1, X_2, \dots , there are two mitigating circumstances. Suppose that the key distribution has a bounded density and that the hash function effectively randomizes the keys. If the number of keys in each partition is large, then the central limit theorem assures us that the relative variation in the number of keys (and hence the number of tuples) in each partition will be small. Furthermore, if we err slightly we can always apply the hybrid hash join recursively, thereby adding an extra pass for the overflow tuples.

3.4. Sort-Merge Join Algorithm

The standard sort-merge algorithm begins by producing sorted runs of tuples which are on the average twice as long as the number of tuples that can fit into a priority queue in memory [KNUT73]. This requires one pass over each relation. During the second phase, the runs are merged using an n -way merge, where n is as large as possible (since only one output page is needed for each run, n can be equal to $|M|-1$). If n is less than the number of runs produced by the first phase, more than two phases will be needed. Our assumptions guarantee that only two phases are needed.

The steps of the sort-merge join algorithm are:

- (1) Scan S and produce output runs using a selection tree or some other priority queue structure. Do the same for R . A typical run will be approximately $\frac{2 * |M|}{F}$ pages long [KNUT73]. Since the runs of R have an average length of $\frac{2 * |M|}{F}$ pages, there are $\frac{|R| * F}{2 * |M|}$ such runs. Similarly, there are $\frac{|S| * F}{2 * |M|}$ runs of S . Since S is the larger relation, the total number of runs is at most $\frac{|S| * F}{|M|}$. Therefore, all the runs can be merged at once if $|M| \geq \frac{|S| * F}{|M|}$, or $|M| \geq \sqrt{|S| * F}$, and we have assumed $|M|$ to be at least $\sqrt{|S| * F}$ pages. Thus all runs can be merged at once.

- (2) Allocate one page of memory for buffer space for each run of R and S. Merge runs from R and S concurrently. When a tuple from R matches one from S, output the pair.

The cost of this algorithm (ignoring the cost of reading the relations initially and the cost of writing the result of the join) is:

$$\begin{aligned}
 & (||R|| \log_2 \frac{\{M\}_R}{F} + ||S|| \log_2 \frac{\{M\}_S}{F}) * (\text{comp} + \text{swap}) && \text{Insert tuples into priority queue} \\
 & && \text{to form initial runs} \\
 & + (|R| + |S|) * IO_{SEQ} && \text{write initial runs} \\
 & + (|R| + |S|) * IO_{RAND} && \text{Reread initial runs} \\
 & + (||R|| \log_2 \frac{||R||}{\{M\}_R/F} + ||S|| \log_2 \frac{||S||}{\{M\}_S/F}) * (\text{comp} + \text{swap}) && \text{Insert tuples into priority queue} \\
 & && \text{for final merge} \\
 & + (||R|| + ||S||) * \text{comp} && \text{Join results of final merge.}
 \end{aligned}$$

This cost formula holds only if a tuple from R does not join with more than a page of tuples from S.

3.5. Simple-Hash Join Algorithm

If a hash table containing all of R fits into memory, i.e. if $|R| * F \leq |M|$, the simple-hash join algorithm proceeds as follows: build a hash table for R in memory and then scan S, hashing each tuple of S and checking for a match with R (to obtain reasonable performance the hash table for R should contain at least TID-key pairs). If the hash table for R will not fit in memory, the simple-hash join algorithm fills memory with a hash table for part of R, then scans S against that hash table, then it continues with another part of R, scans the remainder of S again, etc.

The steps of the simple-hash join algorithm are:

- (1) Let $P = \min(|M|, |R| * F)$. Choose a hash function h and a range of hash values so that $\frac{P}{F}$ pages of R-tuples will hash into that range. Scan the (smaller) relation R and consider each tuple. If the tuple hashes into the chosen range, insert the tuple into a P-page hash table in memory. Otherwise, write the tuple into a new file on disk.
- (2) Scan the larger relation S and consider each tuple. If the tuple hashes into the chosen range, check the hash table of R-tuples in memory for a match and output the pair if a match occurs. Otherwise, write the tuple to disk. Note that if key values of the two relations are distributed similarly, there will be $\frac{P}{F} * \frac{|S|}{|R|}$ pages of the larger relation S processed in this pass.

- (3) Repeat steps (1) and (2), replacing each of the relations R and S by the set of tuples from R and S that were "passed over" and written to disk in the previous pass. The algorithm ends when no tuples from R are passed over.

The algorithm requires $\left\lceil \frac{|R| * F}{|M|} \right\rceil$ passes to execute. We denote this quantity by A. Also note that on

the i th pass, $i = 1, \dots, A-1$, $\|R\| - i * \frac{\{M\}_R}{F}$ tuples of R are passed over. The cost of the algorithm is:

$\ R\ * (\text{hash} + \text{move})$	Place each tuple of R in a hash table
$+ \ S\ * (\text{hash} + \text{comp} * F)$	Check a tuple of S for a match.
$+ ((A-1) * \ R\ - \frac{A * (A-1)}{2} * \frac{\{M\}_R}{F}) * (\text{hash} + \text{move})$	Hash and move passed-over tuples in R.
$+ ((A-1) * \ S\ - \frac{A * (A-1)}{2} * \frac{\{M\}_S}{F}) * (\text{hash} + \text{move})$	Hash and move passed-over tuples in S.
$+ ((A-1) * \ R\ - \frac{A * (A-1)}{2} * \frac{ M }{F}) * 2 * IO_{SEQ}$	Write and read passed-over tuples in R.
$+ ((A-1) * \ S\ - \frac{A * (A-1)}{2} * \frac{ M }{F} * \frac{ S }{ R }) * 2 * IO_{SEQ}$	Write and read passed-over tuples in S

3.6. GRACE-Hash Join Algorithm

As outlined in [KITS83], the GRACE-hash join algorithm executes as two phases. The first phase begins by choosing an h and partitioning the set of hash values for h into $|M|$ sets, corresponding to a partition of R and S into $|M|$ sets each, such that R is partitioned into sets of approximately equal size. No assumptions are made about set sizes in the partition of S. The algorithm uses one page of main memory as an output buffer for each of the $|M|$ sets in the partition of R and S. During the second phase of the algorithm, the join is performed using a hardware sorter to execute a sort-merge algorithm on each pair of sets in the partition. To provide a fair comparison between the different algorithms, we have used hashing to perform the join during the second phase. The algorithm proceeds as follows:

- (1) Scan R. Using h , hash each tuple and place in the appropriate output buffer. When an output buffer fills, it is written to disk. After R has been completely scanned, flush all output buffers to disk.
- (2) Scan S. Using h , hash each tuple and place in the appropriate output buffer. When an output buffer fills, it is written to disk. After S has been completely scanned, flush all output buffers to disk.

Steps (3) and (4) below are repeated for each set R_i , $1 \leq i \leq |M|$, in the partition for R , and its corresponding set S_i .

- (3) Read R_i into memory and build a hash table for it.

We pause to check that a hash table for R_i can fit in memory. Assuming that all the sets R_i are of equal size, since there are $|M|$ of them, $|R_i|$ will equal $\frac{|R|}{|M|}$ pages. The inequality $|R_i| * F \leq |M|$ is equivalent to $\sqrt{|R| * F} \leq |M|$, and we have assumed that $\sqrt{|S| * F} \leq |M|$.

- (4) Hash each tuple of S_i with the same hash function used to build the hash table in (3). Probe for a match. If there is one, output the result tuple, otherwise proceed with then next tuple of S_i .

The cost of this algorithm is:

$(R + S) * (\text{hash} + \text{move})$	Hash tuple and move to output buffer
$+ (R + S) * IO_{RAND}$	Write partitioned relations to disk
$+ (R + S) * IO_{SEQ}$	Read partitioned sets
$+ R * (\text{hash} + \text{move})$	Build hash tables in memory
$+ S * (\text{hash} + \text{comp} * F)$	Probe for a match

3.7. Hybrid-Hash Join Algorithm

In our hybrid-hash algorithm, we use the large main memory to minimize disk traffic. On the first pass, instead of using all of memory as a buffer as is done in the GRACE algorithm, only as many pages (B , defined below) as are necessary to partition R into sets that can fit in memory are used. The rest of memory is used for a hash table that is processed at the same time that R and S are being partitioned.

Let $B = \max(0, \frac{|R| * F - |M|}{|M| - 1})$. There will be $B+1$ steps in the hybrid-hash algorithm.

First, choose a hash function h and partition R into R_0, \dots, R_B , such that a hash table for R_0 has $|M| - B$ pages, and R_1, \dots, R_B are of equal size.

Before describing the algorithm we first show that a hash table for R_i will fit into memory.

Assuming that all sets R_i are of equal size, we denote $|R_i|$ by p . We must show that:

$$p * F \leq |M| \quad (a)$$

Since R_0 is chosen so that a hash table for it fits into $|M| - B$ pages of memory, we have:

$$|R_0| * F = |M| - B \quad (b)$$

Since the sum of all the R_i -sets is R , we have

$$|R| = B * p + |R_0| \quad (c)$$

If a hash table for all of R fits into memory, we can choose $B = 0$ and be done with it. So henceforth we assume $|M| < |R| * F$. Thus, $B = \frac{|R| * F - |M|}{|M| - 1}$. If we solve (c) for p and substitute (b) in the result we get:

$$p = \frac{|R|}{B} - \frac{|R_0|}{B} = \frac{|R|}{B} - \frac{|M| - B}{F * B} \quad (d)$$

Now we multiply (d) by F and simplify to get:

$$p * F = \frac{|R| * F - |M|}{B} + 1 \quad (e)$$

Finally, we substitute for B in (e) to get (a), which was our goal. Thus we have demonstrated that a hash table for R , fits into memory.

Now we continue with the algorithm. Allocate B pages of memory to output buffer space, and assign the other $|M| - B$ pages of memory to a hash table for R_0 . We pause again to check that there are enough pages in memory to hold the output buffers, i.e. that $B \leq |M|$. If we substitute for B in the inequality $B \leq |M|$ and simplify, we get $\sqrt{|R| * F} \leq |M|$, which is true since we have assumed that that $\sqrt{|S| * F} \leq |M|$.

The steps of the hybrid-hash algorithm are:

- (1) Assign the i th output buffer page to R_i for $i = 1, \dots, B$. Scan R . Hash each tuple with h . If it belongs to R_0 , place it in memory in the hash table for R_0 . Otherwise it belongs to R_i for some $i > 0$, so move it to the i th output buffer page. When this step has finished, we have a hash table for R_0 in memory, and R_1, \dots, R_B are on disk. The partition of R corresponds to a partition of S compatible with h , into sets S_0, \dots, S_B .
- (2) Assign the i th output buffer page to S_i for $i = 1, \dots, B$. Scan S , hashing each tuple with h . If the tuple is in S_0 , probe the hash table in memory for a match. If there is a match, output the the result tuple. If there is no match, toss the tuple. Otherwise, the hashed tuple belongs to S_i for some $i > 0$, so move it to the i th output buffer page. Now R_1, \dots, R_B and S_1, \dots, S_B are on disk.

Repeat steps (3) and (4) for $i = 1, \dots, B$.

- (3) Read R_i and build a hash table for it in memory. We have already shown that a hash table for it will fit in memory.

- (4) Scan S_1 , hashing each tuple, and probing the hash table for R_1 , which is in memory. If there is a match, output the result tuple, otherwise toss the S tuple.

For the cost computation, denote by q the quotient $\frac{|R_0|}{|R|}$, namely the fraction of R represented by R_0 .

To calculate the cost of this join we need to know the size of S_0 , and we estimate it to be $q*|S|$. Then the fraction of R and S sets remaining on the disk is $1-q$. The cost of the hybrid-hash join is:

$(R + S)*\text{hash}$	Partition R and S
$+ (R + S)*(1-q)*\text{move}$	Move tuples to output buffers
$+ (R + S)*(1-q)*IO_{RAND}$	Write from output buffers
$+ (R + S)*(1-q)*\text{hash}$	Build hash tables for R and find where to probe for S
$+ S *F*\text{comp}$	Probe for each tuple of S
$+ R *\text{move}$	Move tuples to hash tables for R
$+ (R + S)*(1-q)*IO_{SEQ}$	Read sets into memory

3.8. Comparison of the 4 Join Algorithms

In Figure 1 we have displayed the relative performance of the four join algorithms. The vertical axis is execution time in seconds. The horizontal axis is the ratio of $\frac{|M|}{|R|*F}$. Note that above a ratio of 1.0 all algorithms have the same execution time as at 1.0, except that sort-merge will improve to approximately 900 seconds, since fewer IO operations are needed. The parameter settings used are shown in Table 2. We have assumed that there are at least $\sqrt{|S|*F}$ pages in memory. For the values specified in Table 2, this corresponds to $\frac{|M|}{|R|*F} = 0.009$.

In generating these graphs we have used the cost formulas given above with one exception. The IO_{RAND} term used in the cost formula for hybrid hash should be replaced by IO_{SEQ} in the case that there is only one output buffer. There is only one output buffer whenever $|M| \geq \frac{|R|*F}{2}$ (0.5 on the horizontal axis of Figure 1). The abrupt discontinuity in the performance of the hybrid hash algorithm at 0.5 occurs because when memory space decreases slightly, changing the number of output buffers from one to two, the IO time is suddenly calculated as a multiple of IO_{RAND} instead of IO_{SEQ} . Even when there

Table 2 – Parameter Settings Used

comp	time to compare keys	microseconds
hash	time to hash a key	9 microseconds
move	time to move a tuple	20 microseconds
swap	time to swap two tuples	60 microseconds
IO_{SEQ}	sequential IO operation time	10 milliseconds
IO_{RAND}	random IO operation time	25 milliseconds
F	universal "fudge" factor	1.2
$ S $	size of S relation	10,000 pages
$ R $	size of R relation	10,000 pages
$ R / R $	number of R tuples/page	40
$ S / S $	number of S tuples/page	40

are only two or three buffers, IO_{RAND} is probably too large a figure to use to measure IO cost, but we have not made that change. This is what causes our graphs to indicate that simple hash will outperform hybrid hash in a small region; in practice hybrid hash will probably always outperform simple hash.

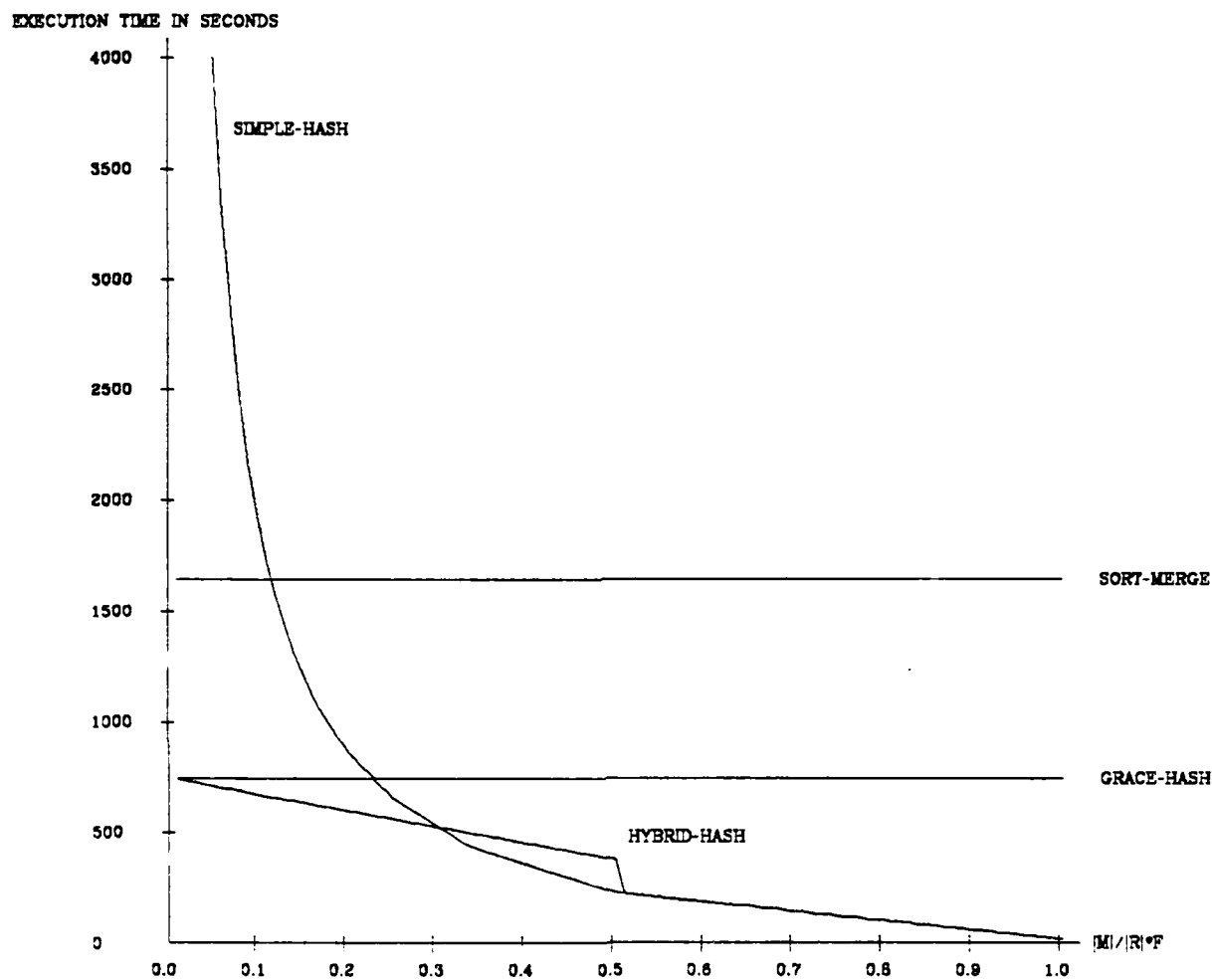
We have generated similar graphs for the range of parameter values shown in Table 3. For each of these values we observed the same qualitative shape and relative positioning of the different algorithms as shown in Figure 1. Thus our conclusions do not appear to depend on the particular parameter values that we have chosen.

3.9. Algorithms for Other Relational Operations

While we have not analyzed algorithms for the remaining relational operations such as aggregate function and projection with duplication elimination, we can offer the following observations. For aggregate functions in which related tuples must be grouped together (compute average employee salary

Table 3 – Other Parameter Settings Tested

comp	1 to 10 microseconds
hash	2 to 50 microseconds
move	10 to 50 microseconds
swap	20 to 250 microseconds
IO_{SEQ}	5 to 10 milliseconds
IO_{RAND}	15 to 35 milliseconds
F	1.0 to 1.4
$ S $	10,000 to 200,000 pages
$ R $	100,000 to 1,000,000 tuples



PERFORMANCE OF THE 4 JOIN ALGORITHMS

Figure 1

by manager), the result relation will contain one tuple per group. If there is enough memory to hold the result relation, then the fastest algorithm will be a one pass hashing algorithm in which each incoming tuple is hashed on the grouping attribute. If there is not enough memory to hold the result relation (probably a very unlikely event as who would ever want to read even a 4 million byte report), then a variant of the hybrid-hash algorithm described for the join operator appears fastest. This same hybrid-hash algorithm appears to be the algorithm of choice for the projection operator as projection with duplicate elimination is very similar in nature to the aggregate function operation (in projection we are grouping identical tuples while in an aggregate function operation we are grouping tuples with an identical partitioning attribute).

4. Access Planning and Query Optimization

In the classic paper on access path selection by Selinger [SEL79], techniques are developed by choosing the "best" processing strategy for a query. "Best" is defined to be the plan that minimizes the function $W \cdot |\text{CPU}| + |\text{I/O}|$ where $|\text{CPU}|$ is the amount of CPU time consumed by a plan, $|\text{I/O}|$ is the number of I/O operations required for a plan, and W is a weighting factor between CPU and I/O resources. Choosing a "best" plan involves enumerating all possible "interesting" orderings of the operators in query, all alternative algorithms for each operator, and all alternative access paths. The process is complicated by the fact the order in which tuples are produced by an operator can have a significant effect on the execution time of the subsequent operator in the query tree.

The analysis presented in Section 3 indicates that algorithms based on hashing (the hybrid-hash algorithm in the case of the join operator and the simple-hash algorithm to process projection and aggregate function operators) are the fastest algorithms when a large amount of primary memory is available. Since the performance of these algorithms is not affected by the input order of the tuples and since there is only one algorithm to choose from, query optimization is reduced to simply ordering the operators so that the most selective operations are pushed towards the bottom of the query tree.

5. Recovery in Large Memory Databases

5.1. Introduction and Assumptions

High transaction processing rates can be obtained on a processor with a large amount of main memory, since input/output delays can be significantly reduced by keeping the database resident in memory. For example, if the entire database is resident in memory, a transaction would never need to access data pages on disk.

However, keeping a large portion of the database in volatile memory presents some unique challenges to the recovery subsystem. The in-memory version of the database may differ significantly from its latest snapshot on disk. A simple recovery scheme would proceed by first reloading the snapshot on disk, and then applying the transaction log to bring it up to date. Unless the recovery system does more than simple logging during normal transaction processing, recovery times would become intolerably long using this approach.

Throughout this section, we will assume that the entire database fits in main memory. In such an environment, we need only be concerned with log writes. A "typical" transaction writes 400 bytes of log data (40 bytes for transaction begin/end, 360 bytes for old values/new values),² which takes 10 ms (time to write one 4096 byte page without a disk seek). We also assume that a small portion of memory can be made stable by providing it with a back-up battery power supply.

5.2. Limits to Transaction Throughput

In conventional logging schemes, a transaction cannot commit until its log commit record has been written to stable storage. Most transactions have very simple application logic, and perform three to four page reads and writes. While transactions no longer need to read or write data pages if the database is memory resident, they still need to perform at least one log I/O. Assuming a single log device, the system could commit at most 100 transactions per second (1 second / 10 ms per commit = 100 committed transactions per second). The time to write the log becomes the major bottleneck.

A scheme that amortizes this log I/O across several transactions is based on the notion of a

² These are ballpark estimates, based on the example banking database and transactions in Jim Gray, "Notes on Database Operating Systems," IBM RJ2188(30001), (February 23, 1978).

pre-committed transaction. When a transaction is ready to complete, the transaction management system places its commit record in the log buffer. The transaction releases all locks *without waiting for the commit record to be written to disk*. The transaction is delayed from committing until its commit record actually appears on disk. The "user" is not notified that the transaction has committed until this event has occurred.³

By releasing its locks before it commits, other transactions can read the pre-committed transaction's dirty data. Call these *dependent transactions*. Reading uncommitted data in this way does not lead to an inconsistent state as long as the pre-committed transaction actually commits *before* its dependent transactions. A pre-committed transaction does not commit only if the system crashes, never because of a user or system induced abort. As long as records are sequentially added to the log, and the pages of the log buffer are written to disk in sequence, a pre-committed transaction will have its commit record on disk before its dependent transactions.

The transactions with commit records on the same log page are committed as a group, and are called the *commit group*. A single log I/O is incurred to commit all transactions within the group. The size of a commit group depends on how many transactions can fit their logs within a unit of log write (i.e., a log buffer page). Assuming the "typical" transaction, we could have up to ten transactions per commit group. The transaction throughput can be increased by another order of magnitude, to 1000 transactions per second (1 second / 10 ms to commit 10 transactions = 1000 transactions committed per second).

The throughput can be further increased by writing more than one log page at a time, by partitioning the log across several devices. Since more than one log I/O can be active simultaneously, the recovery system must maintain a topological ordering among the log pages, so the commit record of a dependent transaction is not written to disk before the commit record of its associated pre-committed transaction. The roots of the topological lattice can be written to disk simultaneously.

To maintain the ordering, and thus the serialization of the transactions, the lock table of the concurrency control component must be extended. Associated with each lock are three sets of transactions: active transactions that currently hold the lock, transactions that are waiting to be granted the

³ The notion of group commits appears to be part of the unwritten database folklore. The System-R implementors claim to have implemented it. To our knowledge, neither the idea nor the implementation details has yet appeared in print.

lock, and pre-committed transactions that have released the lock but have not yet committed. When a transaction is granted a lock, it becomes dependent on the pre-committed transactions that formerly held the lock. The dependency list is maintained in the transaction's descriptor in the active transaction table. When a transaction becomes pre-committed, it moves from the holding list to the pre-committed list for all of its locks (we assume all locks are held until pre-commit), and the committed transactions in its dependency list are removed. In becoming pre-committed, the transaction joins a commit group. The commit groups of the remaining transactions in its dependency list are added to those on which its commit group depends. A commit group cannot be written to disk, and thus commit, until all the groups it depends on have previously been committed.

For recovery processing, a single log is recreated by merging the log fragments, as in a sort-merge. For example, to roll backwards through the log, the most recent log page in each fragment is examined. The page with the most recent timestamp is processed first, it is replaced by the next page in that fragment, and the most recent log page of the group is again determined. By a careful buffering strategy, the reading of log pages from different fragments can be overlapped, thus reducing recovery time.

5.3. Checkpointing the Database

An approach for reducing recovery time is to periodically checkpoint the database to stable storage [GRAY81]. Checkpointing limits recovery activities to those transactions that are active at the checkpoint or who have begun since the last checkpoint. System-R, for example, takes an action consistent checkpoint, during which no storage system operations may be in progress (a transaction consists of several such actions, which correspond roughly to logical reads and writes of the database). Dirty buffer pool pages are forced to disk. Since the database is assumed to be large, a large number of dirty pages will need to be written to disk, making the database unavailable for an intolerably long amount of time. Consider the case of 1000 transactions per second, two dirty pages per transaction, and 30 seconds between checkpoints. In the worst case, 60,000 pages would need to be written at the checkpoint!

We would like to overlap checkpoint with transaction activity. Let Δ_{mem} be the set of pages that have been updated since the last checkpoint. Once a checkpoint begins, transaction activity can continue if updates to pages of Δ_{mem} cause new in-memory versions to be created, leaving the old versions

available to be written to disk. A checkpointed, action consistent state of the database is always maintained on disk. At a checkpoint, a portion of the state is replaced by Δ_{mem} . To guarantee that the state is updated "carefully," we use a batch update approach by first writing these pages to stable storage. We denote the batch update file by Δ_{disk} . If the system crashes while the disk state is being overwritten from memory, it can be reconstructed from the pages in Δ_{disk} .

The algorithm proceeds in two phases. In phase 1, Δ_{mem} is written to Δ_{disk} . During phase 2, the pages in Δ_{mem} are copied to their original locations on disk. For the algorithm to work, we must assume:

- (1) Extra disk space is available to hold Δ_{disk} .
- (2) Extra memory space is available to hold Δ_{mem} .
- (3) No dirty page is ever written to disk except during a checkpoint.

Time stamps are used to determine membership in Δ_{mem} . The timestamp T_{cp} indicates when the current checkpoint began, or is zero if no checkpoint is in progress. When a transaction attempts to update a page, the page's timestamp T_{page} is compared to T_{cp} . If $T_{page} < T_{cp}$ and the page is dirty, a new version of the page is created and the in-core page table points to the new page. The update is applied to the new page. The page's timestamp is updated to reflect the latest modification.

To obtain an action consistent state for the checkpoint, the system is initially quiesced. T_{cp} is set to the current time clock to indicate that a checkpoint has begun. The active transaction list is constructed for later inclusion in the log. Transaction activity can now resume, since the old versions in Δ_{mem} can no longer be updated. Memory pages who are dirty and for which $T_{page} < T_{cp}$ are written to Δ_{disk} . After Δ_{disk} has been created, a *begin checkpoint* record is written to the log with T_{cp} and the list of active transactions, indicating that phase 1 of checkpoint is complete. The pages of Δ_{mem} are then written to their original locations on disk, making the disk state identical to the in-memory state as of T_{cp} . An *end checkpoint* record is written to the log to indicate the completion of phase 2, Δ_{disk} is removed, and T_{cp} is reset.

The advantage of the algorithm is that checkpointing can be done in parallel with transaction activity while maintaining an action consistent state on disk. This is particularly needed in a high update

transaction environment, which can generate a large number of updated pages between checkpoints. Further, as soon as a checkpoint completes, another can commence with only a negligible interruption of service. Checkpointing proceeds at the maximum rate possible, i.e., as fast as pages can be written to disk, thus keeping the log processing time to a minimum during recovery.

5.4. Reducing Log Size

While checkpointing will reduce the time to process the log, by reducing the necessary redo activity, it does not help reduce the log size. The large amount of real memory available to us can be used to reduce the log size, if we assume that a portion of memory can be made stable against system power failures. For example, batteries can be used as a back-up power supply for low power CMOS memory chips. We further assume that such memory is too expensive to be used for all of real memory.

Partition real memory into a stable portion and a conventional, non-stable portion. The stable portion will be used to hold an *in-memory log*, which can be viewed as a reliable disk output queue for log data. Transactions commit as soon as they write their commit records into the in-memory log. Log data is written to disk as soon as a log buffer page fills up. Given the buffering of the log in memory, it may be more efficient to write the log a track at a time. In addition, multiple log writes can be directed to different log devices without the need for the bookkeeping described above. However, in the steady state, the number of transactions processed per second is still limited by how fast we can empty buffer pages by writing them to disk-based stable storage.

Stable memory does not seem to gain much over the group commit mechanism. However, the log can be significantly compressed while it is buffered in stable memory. The log entries for a particular transaction are of the form

```
Begin Transaction
<Old Value, New Value>
. . .
<Old Value, New Value>
End Transaction
```

A transaction's space in the log can be significantly reduced if only new values are written to the disk based log (approximately half of the size of the log stores the old values of modified data — this is only needed if the transaction must be undone). This is advantageous for space management, and also reduces

the recovery time by shortening the log.

In the conventional approach, log entries for all transactions are intermixed in the log. The log manager maintains a list of committed transactions, and removes their old value entries from log pages before writing them to disk. A transaction is removed from the list as soon as its commit record has been written to disk. A more space efficient alternative is to maintain the log on a per transaction basis in the stable memory. If enough space can be set aside to accommodate the logs of all active transactions, then only new values of committed transactions are ever written to disk.

Stable memory also assists in reducing the recovery time. To recover committed updates to pages since the last checkpoint, the recovery system needs to find the log entry of the oldest update that refers to an uncheckpointed page. A table can be placed in stable memory to record which pages have been updated since their last checkpoint, and the log record id of the first operation that updated the page. When a page is checkpointed to disk, its update status is reset. The log record id of the next update on the page is entered into the table. The oldest entry in the table determines the point in the log from which recovery should commence.

6. Conclusions and Future Research

In this paper we have examined changes to the organization of relational database management systems to permit effective utilization of very large main memories. We have shown that the B+ tree access method will remain the preferred access method for keyed access to tuples in a relation unless more than 80% - 90% of the database can be kept in main memory. We have also evaluated alternative algorithms for complex relational operators. We have shown that once the size of main memory exceeds the square root of the size of the relations being processed, that the fastest algorithms for the join, projection, and aggregate operators are based on a hashing. It is interesting to note that this result also holds for "small" main memories and "small" databases. Finally, we have discussed recovery techniques for memory-resident databases.

There appear to be a number of promising areas for future research. These include buffer management strategies (how to efficiently manage very large buffer pools), the effect of virtual memory on query processing algorithms, and concurrency control. While locking is generally accepted to the algo-

rithm of choice for disk resident databases, a versioning mechanism [REED83] may provide superior performance for memory resident systems.

7. References

- [BABB79] Babb, E. "Implementing a Relational Database by Means of Specialized Hardware," ACM TODS, Vol. 4, No. 1, March 1979.
- [BLAS77] Blasgen, M.W. and K.P. Eswaran, "Storage and Access in Relational Databases," IBM Systems Journal, Vol. 16, No. 4, 1977.
- [CESA82] Cesarini, F. and G. Soda, "Binary Trees Paging", Information Systems, Vol. 7, No. 4, pp 337-344, 1982.
- [COME79] Comer, D., "The Ubiquitous B-tree," ACM Computing Surveys, Vol. 11, No. 2, June 1979.
- [DATE82] Date, C.J., "An Introduction to Database Systems," Third Edition, Addison-Wesley, 1982.
- [GOOD81] Goodman, J. R., "An Investigation of Multiprocessor Structures and Algorithms for Data Base Management," Electronics Research Laboratory Memorandum No. UCB/ERL M81/33, University of California, Berkeley, May 1981.
- [GRAY81] Gray, J., et. al., "The Recovery Manager of the System R Database Manager," ACM Computing Surveys, Vol. 13, No. 2, June 1981.
- [KNUT73] Knuth, D., "The Art of Computer Programming: Sorting and Searching." Volume III, Addison-Wesley, Reading, MA, 1973.
- [KITS83] Kitsuregawa, M. et al, "Application of Hash to Data Base Machine and its Architecture", New Generation Computing, No. 1, 1983, 62-74.
- [MUNT70] Muntz, R. and R. Uzgalis, "Dynamic Storage Allocation for Binary Search Trees in a Two-Level Memory," Proceedings of the Princeton Conference on Information Sciences and Systems, No. 4, pp. 345-349, 1970.
- [REED83] Reed, D., "Implementing Atomic Actions on Decentralized Data," ACM Transactions on Computer Systems, V 1, N 1, (March 1983).
- [SELI79] Selinger, P.G., et. al., "Access Path Selection in a Relational DBMS," Proceedings of the 1979 SIGMOD Conference on the Management of Data, June 1979.
- [YAO78] Yao, S. B., and D. DeJong, "Evaluation of Database Access Paths," Proceedings of the 1978 SIGMOD Conference on the Management of Data, May 1978.